The Dissertation Committee for Jian Chen
certifies that this is the approved version of the following dissertation:

Resource Management for Efficient Single-ISA Heterogeneous Computing

Committee:

Lizy Kurian John, Supervisor

Earl E. Swartzlander, Jr.

Joydeep Ghosh

David Z. Pan

Lieven Eeckhout

Resource Management for Efficient Single-ISA Heterogeneous Computing

by

Jian Chen, B.E.; M.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

Dedicated to my parents, Zelin Hu and Yuehuai Chen.

Acknowledgements

I am grateful for many people who helped me throughout my journey of PhD study, and I would like to take this opportunity to acknowledge their efforts.

First of all, I would like to thank my advisor, Lizy K. John, for providing me with the freedom and the resources to do high-quality research and for teaching me valuable lessons in research and beyond.

I would also like to thank all the members of LCA group. I especially thank Dimitris Kaseridis for establishing the simulation platform and making the group a fun place to work at, Arun Nair for the wonderful collaboration, Ciji Isen for bringing some software aspects in my research, Jeff Stuecheli for sharing his industry perspectives, and Karthik Ganesan, Muhammad U. Farooq, Faisal Iqbal, Jungho Jo, YoungTaek Kim for their valuable feedbacks on my papers and presentations.

Very special thanks to Dong Li for the intelligent discussions on research as well as the sincere sharing of life stories, which made the stressful PhD life a bit easier.

Many thanks to Lieven Eeckhout, Earl Swartzlander, Joydeep Gosh and David Pan for serving in my PhD committee.

Finally, I would like to thank my parents, Zelin Hu and Yuehuai Chen, my girlfriend, Shifang Liu, for their unconditional support throughout my PhD study. Without their encouragement and dedication, I would not have completed this dissertation.

Resource Management for Efficient Single-ISA Heterogeneous Computing

Publication No._____

Jian Chen, Ph.D.

The University of Texas at Austin, 2011

Supervisor: Lizy Kurian John

Single-ISA Heterogeneous Multi-core Processors (SHMPs) have become increasingly important due to their potential to significantly improve the execution efficiency for diverse workloads and thereby alleviate the power density constraints in Chip Multiprocessors (CMPs). The importance of SHMP is further underscored by the fact that manufacturing defects and process variation could also cause single-ISA heterogeneity in CMPs even though the CMP is originally designed as homogeneous. However, to fully exploit the execution efficiency that SHMP has to offer, programs have to be efficiently mapped/scheduled to the appropriate cores such that the hardware resources of the cores match the resource demands of the programs, which is challenging and remains an open problem.

This dissertation presents a comprehensive set of off-line and on-line techniques that leverage analytical performance modeling to bridge the gap between the workload diversity and the hardware heterogeneity. For the off-line scenario, this dissertation presents an efficient resource demand analysis framework that can estimate the resource

demands of a program based on the inherent characteristics of the program without using any detailed simulation. Based on the estimated resource demands, this dissertation further proposes a multi-dimensional program-core matching technique that projects program resource demands and core configurations to a unified multi-dimensional space, and uses the weighted Euclidean distance between these two to identify the matching program-core pair.

This dissertation also presents a dynamic and predictive application scheduler for SHMPs. It uses a set of hardware-efficient online profilers and an analytical performance model to simultaneously predict the application's performance on different cores. Based on the predicted performance, the scheduler identifies and enforces near-optimal application assignment for each scheduling interval without any trial runs or off-line profiling. Using only a few kilo-bytes of extra hardware, the proposed heterogeneity-aware scheduler improves the weighted speedup by 11.3% compared with the commodity OpenSolaris scheduler and by 6.8% compared with the best known research scheduler.

Finally, this dissertation presents a predictive yet cost effective mechanism to manage intra-core and/or inter-core resources in dynamic SHMPs. It also uses a set of hardware-efficient online profilers and an analytical performance model to predict the application's performance with different resource allocations. Based on the predicted performance, the resource allocator identifies and enforces near optimum resource partitions for each epoch without any trial runs. The experimental results show that the proposed predictive resource management framework could improve the weighted speedup of the CMP system by an average of 11.6% compared with the equal partition scheme, and 9.3% compared with existing reactive resource management scheme.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chip Multiprocessors (CMP) have become the mainstream computing platform to improve the utilization of the abundant yet ever-increasing on-chip resources and alleviate the power density constraints. By integrating multiple cores in a single chip, CMP allows multiple programs or multiple threads to simultaneously execute on different cores on the same chip, significantly improving the system throughput and efficiency. However, CMPs composed of *homogeneous* processor cores still suffer from inefficiency because they lead to an inevitable dilemma: replicating smaller cores compromises the throughput of the high-complexity single-threaded applications; whereas replicating larger cores sacrifices the execution efficiency of the low-complexity low-priority applications. Therefore, to achieve high efficiency, CMPs need certain amount of core-level heterogeneity to accommodate or adapt to the diverse workload requirements.

1.1 Single-ISA Heterogeneous Multi-core Processor

Single-ISA Heterogeneous Multi-core Processor (SHMP) [1] emerges as an important and attractive type of CMP that provides the core-level heterogeneity to meet the diverse requirements of the workloads. It consists of cores with the same Instruction-Set Architecture (ISA) yet different configurations, and hence allows any application/task to be executed on any core in the system without modification in the

application binaries. Depending on the runtime configurability of the processor cores, SHMP can be categorized as *Static Single-ISA Heterogeneous Multi-core Processor* or *Dynamic Single-ISA Heterogeneous Multi-core Processor*.

Static SHMP is statically composed of cores with fixed yet different configurations. Such static heterogeneity can be introduced intentionally by integrating cores with different complexity in a single chip at the design stage or can be caused unintentionally by process variation and hardware defects during the manufacturing stage. Either way, it relies on an appropriate scheduling scheme to map the program to the processor core that matches the program's resource demands [2][3]. On the other hand, dynamic SHMP is realized by dynamically reconfiguring the cores as well as other on-chip resources, such as L2 cache sizes, to meet the need of the applications. Examples of such SHMP include Tflex [4] and Core Fusion [5]. Such SHMPs are able to meet the application resource demand changes at a finer granularity during runtime, yet their hardware adaptation still relies on the workload heterogeneity being properly translated to the corresponding hardware resource demands.

1.2 The Problem: Gap between Workload Diversity and Hardware Heterogeneity

In static SHMPs, mapping the workload diversity to the hardware heterogeneity is achieved by heterogeneity-aware application scheduling; whereas in dynamic SHMPs, it is typically achieved by runtime resource adaptation. However, in both cases, the existing methods are severely constrained by their inefficiency, poor scalability, and inability to enforce performance objectives.

Issues on Efficiency: The conventional approach for heterogeneity-aware

application scheduling relies on tentative runs to explore the appropriate application-core mapping [2][6]. Specifically, this approach tentatively executes the application on different cores, each for a short period of time, and then schedules the program to the optimum core based on the sampled Energy-Delay Product (EDP) during the tentative runs. This method suffers inefficiency from the trial runs as they incur significant power and performance overhead in moving around the architecture states and data sets, potentially negating the benefit of the improvement in application scheduling. Similarly, the conventional method for dynamic resource adaptation also uses the trial-and-error approach to explore the appropriate resource allocation, which also causes inefficiency because a large amount of execution time is spent in exploring the trial resource allocations.

Issues on Scalability: The trial-and-error application scheduling in static SHMPs is only feasible for a relatively small number of cores, because the number of trial runs for exploring all scheduling options grows almost exponentially as the number of heterogeneous cores increases. Similarly, for dynamic resource adaptation, the time needed for tentative runs becomes almost intractable for a large reconfiguration space, resulting in slow adaptation and poor performance.

Issues on Performance Objective Enforcement: Providing performance isolation for the co-executing applications and enforcing the given performance objectives are becoming increasingly important as CMPs expand their usage toward service-oriented computing, server consolidation and virtualization [7]. However, dynamic resource adaptation using tentative runs takes long latency to find the appropriate resource allocations for the system performance objectives. Such a long response latency may occupy a significant portion of the program phase, or even

exceed the phase duration, resulting in inefficiency or even inability to provide performance level guarantees.

As a result, to unleash the full potential for heterogeneous computing, there is an urgent need of efficient and scalable techniques that can translate the workload diversity to the hardware heterogeneity, and enforce system performance objectives. To this end, this dissertation presents and evaluates a set of techniques for analyzing the resource demands and thereby managing multiple interactive resources.

1.3 Proposed Approach

Whether it is application scheduling in static SHMP or resource management in dynamic SHMP, efficiently identifying the resource demand of the application is the key step to close the gap between the workload heterogeneity and the hardware heterogeneity. To do so, this dissertation proposes to leverage analytical performance modeling and micro-architecture independent characteristics for resource demand identification, as shown in Figure 1.1. Unlike the regression models [8][9] or the neural network models [10], the analytical performance model does not require training which significantly boosts the efficiency in exploring the configuration space both off-line and on-line. Using micro-architecture independent characteristics also allows the resource demand identification to be isolated from any partial simulations, not only speeding-up the process of resource demand analysis, but also enabling on-line performance prediction.

Based on the identified resource demands, this dissertation presents a multi-dimensional matching technique [11] for off-line program-core mapping, and a performance prediction scheme for on-line application scheduling to improve the computing

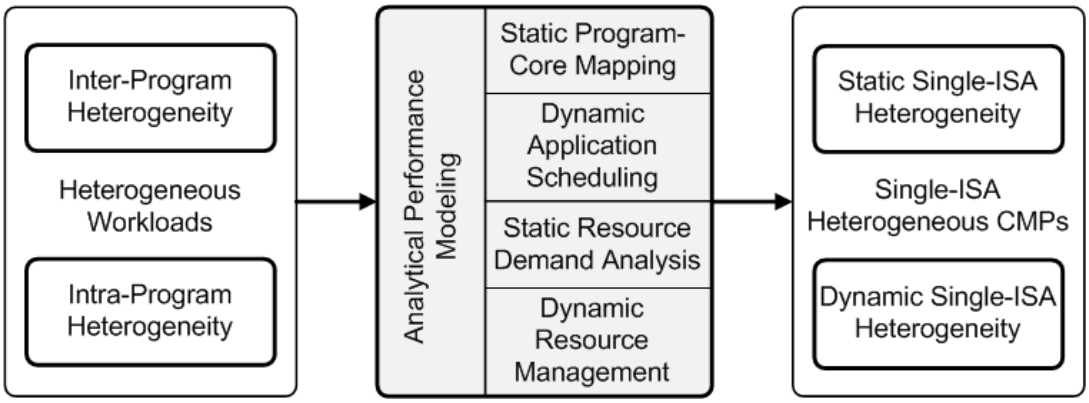


Figure 1.1: Overview of the proposed research.The grey boxes highlight the scope of this research.

efficiency in static SHMP. On the other hand, using the proposed analytical model, this dissertation also presents a comprehensive yet cost-effective resource management framework that can coordinate multiple shared resources while simultaneously enforcing Quality-of-Service (QoS) performance objectives.

1.4 Thesis Statement

The combination of analytical performance modeling and micro-architecture independent characteristics provides an attractive platform for program resource demand analysis, which further enables efficient and scalable application scheduling and multiple resource management for the single-ISA heterogeneous computing environment.

1.5 Contributions

This dissertation makes the following major contributions:

1. This dissertation presents an analytical model based on program character-

istics such as Instruction Level Parallelism (ILP), Memory Level Parallelism (MLP) and branch predictability. Unlike existing analytical models [12][13], which require simulations on caches and branch predictors, this proposed model avoids any partial detailed simulation; yet it is still able to accurately model the performance trend for different hardware configurations. The decoupling from detailed simulation not only makes this model efficient in estimating the resource demand off-line, but also allows it to be applied in proactive on-line resource management. This dissertation also encapsulates the analytical model and the resource demand estimation algorithms into an integrated framework called *Program REsource Demand Analyzer* (PREDA), which automatically estimates a broad set of resource demands for a workload. Compared with the framework using a state-of-the-art analytical model [13], this framework achieves significant speedup in estimating multi-resource demands.

2. This dissertation presents an off-line program-core mapping framework for static heterogeneous CMPs. The proposed framework projects the core configurations and the programs resource demands into a unified multi-dimensional space, where the program-core matching can be easily identified with weighted Euclidean distances. This dissertation demonstrates that the weighted Euclidean distance is strongly correlated with EDP, hence can be used to guide program scheduling in heterogeneous multicores.

3. This dissertation builds a comprehensive yet cost-effective dynamic on-line profiler, and a performance predictor that utilizes the online profile to accurately predict the performance of cores with different configurations on multiple resources. Based on the proposed performance predictor, this dissertation further

proposes a framework for dynamic heterogeneity-aware application scheduling. This scheduling framework eliminates the need of trial-runs or off-line profiling, yet can dynamically and efficiently adapt to program phases. Experimental results show that the proposed approach significantly improves the throughput and efficiency compared with the commodity *OpenSolaris* scheduler and with the best known research scheduler.

4. This dissertation presents a framework for multiple resource management based on the proposed performance model. By using a set of on-line profilers, the performance model is able to predict the performance of the running applications for different resource allocations of both inter-core an intra-core resources, and hence fundamentally eliminating the need of trial-runs or training required for conventional dynamic resource management schemes. This framework is also able to efficiently translate system performance specification to the resource usages, hence it allows the enforcement of QoS performance objectives when multiple interacting resources are reconfigured.

1.6 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 gives background and the related work in this area. Chapter 3 presents the details of the analytical performance model. Chapter 4 describes the experiment platforms, workloads as well as the metrics employed in this dissertation. Chapter 5 shows the mechanism for estimating program resource demands using the proposed performance model. Chapter 6 demonstrates the framework of off-line program-core mapping in static SHMP. Chapter 7 presents the framework for dynamic and predictive application

scheduling in static SHMP. Chapter 8 shows the mechanism for predictive coordination of multiple interacting resources in dynamic SHMP. Chapter 9 summarizes of the key results and insights presented in the dissertation, and proposes the directions for future research.

Chapter 2

Background and Related Work

2.1 Related Research on Creating SHMP

Single-ISA heterogeneous multi-core architecture was initially proposed by Kumar, *et al.* [2] as an attractive option for power-efficient computing. Their proposed heterogeneous multicore processor is composed of legacy Alpha cores with different complexities, effectively amortizing the design and verification effort. IBM CELL processor is another variant of heterogeneous multicore processor, which combines a PowerPC Core with eight Synergistic Processing Elements (SPE) [14]. Although the core and SPE uses different instruction sets, the CELL processor does underscore the importance of using heterogeneity or hardware specialization to boost system efficiency.

Besides the intentionally introduced hardware heterogeneity, single-ISA heterogeneity can also be unintentionally introduced by process variation and manufacturing defects. Process variation is defined as a divergence in the parameters of the fabricated transistors from their nominal values, both within dies (WID) and die-to-die (D2D) [15]. It occurs due to random dopant fluctuations and shortcomings of lithographic processes, and could significantly affect the threshold voltage of transistors. ITRS [16] reports that the 3σ intra-die variation of a transistor's threshold voltage and effective channel length can be as large as 42% and 12% in 45nm technology, and is expected to be worsen as the technology scales down further. The

variation on these parameters directly impacts the switching speed of the transistors, which further causes the maximum operating frequency of the processor to deviate from its nominal value. In a multi-core processor, this implies that different cores may need to operate at different maximum frequencies.

Besides process variation, hard faults are another important issue in manufacturing process. They are caused by imprecise calibration of equipment, contaminants in materials, as well as particle impurities in the air [17], and could incur functional failures in parts of the processors, resulting in expensive yield loss. It is expected that the yield loss will be exacerbated as the transistor density and die size increase, and needs to be carefully controlled. To mitigate yield loss, industry typically leverages the redundancy in processor components such as SRAM arrays, functional units and queues, and recovers faulty processors by disabling some of the defective yet non-critical units [17]. These rescued processors are fully functional, albeit with reduced performance due to the reduction in certain hardware resources. That said, not all faulty units are suitable for this yield-enhancing technique: faults in critical units, such as control units, could cause complete failure of the processor, and faults in Reorder Buffer (ROB) or load/store queue, may require complex and expensive hardware to recover the functionality. Hence, this dissertation focuses on two types of representative resources that can be protected by this yield-enhancing technique, namely, available functional units and L2 cache size. Functional units have their natural redundancy in microprocessors, especially in wide-issue superscalar processors, and have been explored to improve the yield [17]. L2 cache occupies a large amount of chip real estate, and is susceptible to hard faults. While it is typically equipped with redundancy to improve the yield, the occurrence of hard faults may exceed the protection capability. Should this happen, the defective ways in L2 cache can be

discovered and disabled during manufacturing test, which results in a smaller, but functional cache.

While the above heterogeneous CMP are static, the singe-ISA heterogeneity in CMP can also be formulated dynamically. Kim, *et al.* proposed the Tflex architecture [4], which allows simple and low-power cores to be aggregated together dynamically, creating larger, more powerful processors without changing the application binary. Ipek, *et al.* also present a flexible chip multiprocessor substrate that can dynamically morph multiple cores into a larger processing unit [5] to meet the runtime requirements of the applications.

2.2 Related Research on Application Scheduling in Static SHMP

Along the proposal of SHMP, Kumar, *et al.* [2] also propose a dynamic program scheduling approach based on the sampled EDP during tentative runs. This method tentatively runs the program on different cores, each for a short period of time, and then schedules the program to the optimum core according to the sampled EDP during the tentative runs. The downside of this method is the expensive context switching cost of the tentative runs, which may significantly degrade the overall efficiency of the multi-core system. Becchi, *et al.* extends Kumar's work by measuring IPC ratios between two different cores to migrate applications [6]. Essentially, this method uses pair-wise program swapping to reduce the number of trial executions. Nevertheless, this method still relies on tentative runs to identify the matching program-core pair, hence suffers from the same limitation as Kumar's method. Beside the extra power overhead, such trial-and-error approaches also incur scalability issues as the number of cores increases. In future many-core chips, sam-

pling a large amount of cores before scheduling the program would be impractical because the extra cost of sampling will exceed the potential gain of core switching.

In contrast to dynamic scheduling, Chen and John [18] present an off-line technique that leverages the inherent program characteristics for the static program mapping. They employ fuzzy logic to calculate the program-core suitability, and use that to guide program scheduling. However, their method is not scalable since the complexity of fuzzy logic increases exponentially as the number of characteristics increases. Recently, the idea of using program characteristics to guide program scheduling was also employed by Shelepov, *et al.* in their heterogeneity-aware scheduler [3]. Their proposal utilizes reuse distance signatures constructed from off-line profiling to schedule applications to cores with different cache sizes. These off-line approaches can only schedule the application statically, missing opportunities for exploiting changes in program phase. Moreover, the need for off-line profiling and encoding/decoding of the profiled information in the program binary could result in dramatic modification in the interface between OS, compiler and architecture, which limits the applicability of the off-line approach in practice.

2.3 Related Research on Dynamic Resource Management

The dynamic SHMP relies on dynamic resource management techniques to detect the resource demand changes and manage the hardware resource accordingly. These techniques usually leverage hardware performance counters to monitor the performance variations on the fly [19] and *reactively* tune the hardware resources until it meets the demand of the workload [20]. Choi and Yeung [21] improve the SMT resource distribution by *directly* using the performance feedback to partition

the resources for a specific performance goal. Their method requires a number of trial resource partitions before it learns the appropriate resource distribution, fundamentally limiting its potential for performance improvement. The trial-and-error nature of the process may require many tuning iterations, and could incur significant performance degradation and energy overhead.

Recently, there are some *proactive* schemes proposed. Cazorla , *et al.* [22] proposed a DCRA mechanism to dynamically allocate shared resources to each thread in an SMT processor. Their method uses a resource sharing model to estimate the thread's anticipated resource needs, and allocate resources to the thread that utilizes the resource most efficiently. Like other SMT resource sharing policies [23][24], this method improves the SMT performance only *indirectly*, not only potentially missing opportunities for further performance improvement but also unable to control the end performance. Qureshi, *et al.* proposed cache utility monitor (UMON) to estimate the utility of assigning additional cache ways to an application [25]. Kaseridis, *et al.* extended this on-line cache monitor for system-level memory bandwidth management [26]. While these works address single resource management, Bitirgen, *et al.* attempted to manage multiple resources by using on-line machine learning techniques [27]. However, the on-line machine learning model requires periodic training and is hard to implement and validate. In contrast, our proposed model does not require any training and could be applied on-line for both single or multiple resource management.

2.4 Related Research on Performance Modeling

Whether it is application scheduling in static SHMP or resource management in dynamic SHMP, accurate and efficient performance modeling is the key step to identify the program resource demand changes and thereby make adjustments to achieve efficient heterogeneous computing. The performance modeling usually employs analytical models, regression models, or predictive models.

The analytical model is typically based on *interval analysis*, which was used by Karkhanis and Smith for their first-order superscalar processor model [12]. They further leveraged this model to automatically explore the design space for the Pareto-optimal design parameters [28]. Recently, this model was improved by Eyerman, *et al.* for a higher accuracy in performance modeling [13]. However, all of these models rely on detailed simulation of some components, such as caches and branch predictors, to obtain key statistics of the program-microarchitecture interaction. The requirement for partial simulation not only costs time in off-line performance modeling, but also implies that it has to follow the trial-and-error scheme when applying this model for on-line resource management. However, our approach focuses on modeling the performance trend rather than the absolute performance value, and avoids any detailed simulation of any resource component. The decoupling from detailed simulations not only ensures fast off-line resource demand estimation, but also allows this model to be applied in *proactive* on-line resource management.

Both regression models and predictive models are essentially empirical models, which hide the details of program-hardware interactions by fitting high-level equations with the simulated results. The regression model has been applied in estimating the significance of the design parameters and their interactions [8], exploring

the design space [9] as well as analyzing the microarchitectural adaptivity [29]. An artificial neural network (ANN) based predictive model was also proposed by Ipek, *et al.* for performance prediction [10]. While these empirical models are relatively simple, they require time consuming training on a per-application basis before they can model the performance with reasonable accuracy. The requirement for training fundamentally limits these models from being applied on-line.

Chapter 3

Analytical Performance Modeling

Efficient resource demand identification requires fast feedback of the performance under different resource allocations. To meet this requirement, this dissertation employs an analytical model to estimate the application's performance in the searching process of resource demands. Compared with regression models or machine learning models, the analytical model is simple and does not require any training, hence it is a good candidate for resource demand identification. This chapter explains the details of the basic and the extended analytical performance model that will be used in the following chapters for resource demand analysis, heterogeneity-aware application scheduling, and dynamic resource management.

3.1 Basic Analytical Model

The analytical performance model is based on the previously proposed interval analysis [12][19], which treats the exhibited Cycle-Per-Instruction (CPI) rate as a sustained steady state execution rate intermittently disrupted by long-latency miss events, such as, L2 cache misses and branch misprediction, etc. With the interval analysis, the total CPI of an application can be treated as the sum of three CPI components [30]: $CPI_{total} = CPI_{exe} + CPI_{mem} + CPI_{other}$.

CPI_{exe} represents the steady-state execution rate when the execution is free from any miss events. It is fundamentally constrained by the Instruction Level Par-

allelism (ILP) of the application and the issue width of the processor. The ILP of the application is typically characterized by the critical dependency chain of the instructions in the *instruction window* (equivalent to *reorder buffer* in this dissertation). Assume an instruction window size w, and average critical dependency chain length l_w. On an idealized machine with unit execution latency, l_w indicates the average number of cycles required to execute the instructions in the instruction window, hence the average throughput is w/l_w. For a more realistic machine with non-unit execution latency, this number should be further divided by the average execution latency lat_{avg} according to Little's law [12]. Therefore, the average ILP, α_{avg}, can be obtained by $w/(lat_{avg} \cdot l_w)$, which also represents the steady-state execution rate if the instruction issue width is unlimited. However, for a realistic processor with limited issue width β, the steady-state execution rate would be saturated at either the average ILP or the issue width, whichever is smaller. As a result, CPI_{exe} can be obtained by $1/min(\alpha_{avg}, \beta)$.

CPI_{mem} represents the penalty caused by the load misses in the last level cache (L2 cache in this paper). It can be calculated by the multiplication between the number of L2 load misses N_{L2}, and the average memory access latency lat_{mem}, assuming there are no multiple L2 cache misses outstanding. In practice, in order to hide the load miss latency, L2 caches are usually non-blocking and multiple L2 cache load misses could be outstanding. Under these circumstances, it has been proven that the average load miss latency is reduced to lat_{mem}/m_{ovp} [12], where m_{ovp} is the average number of outstanding load misses. Therefore, CPI_{mem} can be calculated by $lat_{mem} \cdot N_{L2}/(m_{ovp} \cdot N_{inst})$, where N_{inst} is the total number of retired instructions. Note that the term N_{L2}/m_{ovp} could also be treated as the number of L2 load misses that are not overlapping with each other, and hence is referred to as

the *non-overlapped L2 load misses* N_{novp}.

$$CPI_{total} = \frac{1}{min(\alpha_{avg}, \beta)} + \frac{lat_{mem} \cdot N_{novp}}{N_{inst}} + CPI_{other}$$

CPI_{other} is the CPI component caused by other miss events, such as instruction cache misses, branch mispredictions, etc. This dissertation assumes that the resources related with these miss events remain unchanged for different cores. Therefore, this CPI component can be treated as a constant parameter when an application is migrated from one core to anther as long as the application is in stable execution phase. The value of this CPI component can be obtained by transforming equation (1) to $CPI_{other} = CPI_{total} - CPI_{exe} - CPI_{mem}$, where CPI_{total} can be obtained from the performance counter, CPI_{ideal} and CPI_{mem} can be derived from the observed program characteristics. The deduced CPI_{other} can then be plugged into the analytical model to estimate the performance of other cores. Therefore, the basic performance model can be written as follows:

3.2 Extended Performance Model

The basic performance model itself is unable to predict the performance of processor cores with different resource configurations. Specifically, it assumes that each core has a sufficient number of functional units (FUs), the same operating frequency, the same L2 cache sizes, and is executing in single-thread mode. However, when the number of functional units is limited, instructions may be stalled for additional cycles, resulting in lower performance. Similarly, different L2 cache sizes could also influence the number of non-overlapped L2 load misses, and if the core supports Simultaneous Multi-threading (SMT), the co-executing thread(s) could also

influence a thread's performance. Therefore, in order to estimate the performance of different resource allocations, the basic analytical model has to be augmented to capture the performance impact of limited functional units, different operating frequencies, different L2 cache sizes, and the interaction of co-executing threads.

3.2.1 Impact of Limited Functional Units

The limited functional units may stall the instructions for additional cycles, which impacts the performance from two aspects. First, the additional stalled cycles increase the average execution latency, which in turn reduces the observed average ILP. Second, the limited number of functional units may also constrain the number of the instructions that can be issued in one cycle, causing the effective issue width β_{eff} smaller than the nominal one.

To capture these performance impacts, this dissertation presents the *ready set size histogram* for any given type of FU. The *ready set* is the set of instructions in the instruction window that are ready for execution on a certain type of functional units, and the *ready set size* (RSS) is the number of instructions in the ready set, used as an index to the *ready set size histogram*. Each time a new ready set is encountered, the histogram entry indexed with the corresponding RSS is incremented by one. As shown in Figure 3.1(a), when instruction a finishes execution, instructions b and c are ready to execute. Since both b and c will execute on an Integer ALU (I-ALU), the RSS for I-ALU is 2 and the corresponding entry in the I-ALU RSS histogram is incremented. When instruction b finishes execution, instructions d, e and f are free. Instruction d will execute on load unit; both e and f will execute on the I-ALU, though they have different opcodes. Hence, the new RSS for I-ALU is also 2. Note that even if at this point c is still in ready state, it should not be counted in the new

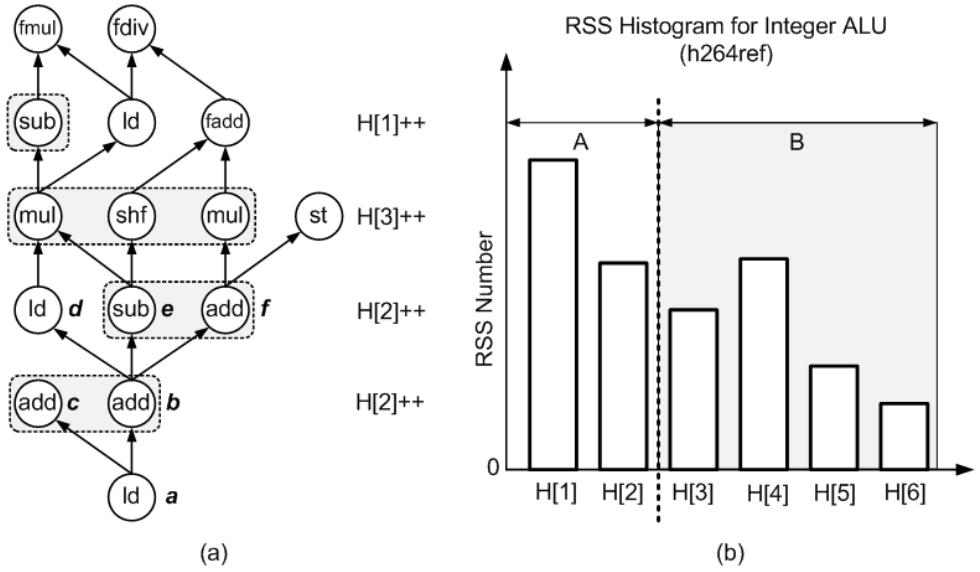


Figure 3.1: The instruction ready set and the RSS histogram.(a) Example of an instruction dependency graph. (b) I-ALU RSS histogram for SPEC CPU2006 program *h264ref*.

ready set. Therefore, RSS histogram reflects the inherent property of the workload, and is microarchitecture-independent.

The RSS histogram opens the door to estimate the number of stalled cycles and the effective issue width for any number of FUs. As shown in Figure 3.1(b), the number of I-ALU splits the histogram into two regions. Region A contains instructions with RSS no larger than the I-ALU number, hence instructions in this region would not experience additional stalled cycles caused by I-ALU. While in region B, the I-ALU number is smaller than RSS, causing additional waiting cycles on the ready instructions. Assuming n fully pipelined I-ALU and a ready set with RSS of m, it takes $\lfloor m/n \rfloor$ additional cycles to finish issuing the instructions in this ready set, contributing an additional cycle-instruction product $\sum_{i=1}^{\lfloor m/n \rfloor}(m - i \cdot n)$ to the equation of calculating the average instruction latency. Therefore, by considering all the additional stalled cycles caused by a limited number of FUs, the average instruction

latency may be changed significantly, resulting in a modified observed average ILP, which is referred to as α'_{avg} in this dissertation. On the other hand, instructions in region A and instructions in region B have different observed issue width. While the observed issue width for the instructions in region A equals the physical issue width, the observed issue width for those in region B is limited by the FU number n. Therefore, on average, the effective issue width $\beta_{eff} = pn + (1-p)\beta$, where p is the percentage of instructions in region B among the total instructions executed. As a result, with the limited functional units, CPI_{exe} becomes $1/min(\alpha'_{avg}, \beta_{eff})$.

3.2.2 Impact of Operating Frequency

Besides modeling the impact of limited FU numbers, the basic performance model also needs to be augmented to capture the performance impact of different clock frequencies. This could be achieved by converting the CPI to the delay in terms of absolute execution time. Consequently, the extended performance model that considers both operating frequency and limited number of FUs can be written as follows:

$$\begin{aligned} Delay &= CPI_{total} * N_{inst}/f \\ &= \frac{N_{inst}}{min(\alpha'_{avg}, \beta_{eff}) \cdot f} + t_{mem} \cdot N_{novp} + C_{other}/f \end{aligned}$$

where N_{inst} is the total number of instructions, f is the operating frequency, t_{mem} represents the absolute memory access time, and C_{other} refers to the product of CPI_{other} and N_{inst}.

3.2.3 Impact of L2 Cache Size

The L2 cache size has direct impact on the number of L2 load misses, which in turn influences the non-overlapped L2 load misses. To estimate the number of L2 load misses, this dissertation employs Mattson's stack distance model [31]. This model exploits the inclusion property of Least Recently Used (LRU) replacement policy (i.e., the content of an N sized cache is a subset of the content of any cache larger than N) and allows us to accurately estimate the number of misses in any fully associative cache. Specifically, this model treats the cache as a large stack organized from Most Recently Used (MRU) position to the LRU position, and each time when a data block is reused, the distance between the position of the block and the MRU position is referred as the *stack distance* of the block. When a load/store accesses a data block with the stack distance larger than the given cache size, that load/store triggers a cache miss event in a fully associative cache. When it comes to set-associative caches, however, the accuracy of this model slightly decreases mainly because it is unable to capture the conflict misses.

While the block-level stack distance can be profiled off-line, the on-line stack distance profiler is implemented at the cache way level since way partition is more feasible and efficient in terms of implementation cost [25]. As an example, Figure 3.2 shows the stack distance histogram of program *xalancbmk* on an 8-way associative cache, organized from MRU position to LRU position. For caches with its associativity reduced to 6-ways (dash line in the figure), the data with stack distance larger than 6 cannot be hold in the cache, generating cache misses. Therefore, with the stack distance histogram, one can estimate the cache miss rate for any cache ways less than the profiled ways and consequently derive the number of L2 misses.

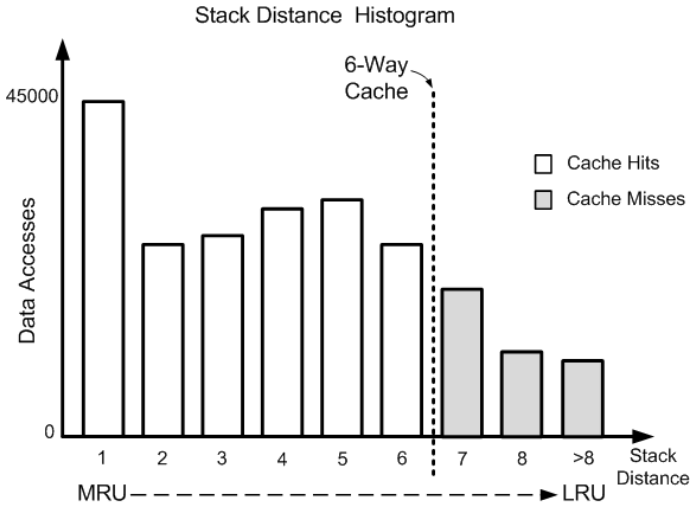


Figure 3.2: Stack Distance Histogram of SPEC CPU2006 program *xalancbmk*.

3.2.4 Impact of Memory-Level Parallelism

While the stack distance model is able to estimate the number of misses for a given cache size, it is unaware of Memory-Level Parallelism (MLP), i.e., multiple L2 load misses overlapping with each other. In the basic performance model, the MLP is modeled by using the average number of outstanding load misses m_{ovp}. However, for a given application, this number can be affected by two factors: the L2 cache size, which determines the total number of L2 load misses; the ROB size, which imposes a "window" on the dynamic instruction stream and controls the amount of exposed MLP. Therefore, to estimate the number of non-overlapped L2 load misses for different ROB sizes and L2 cache sizes, the compounded effect of these two has to be carefully modeled.

Prior research obtains the program's MLP information by simulating caches in detail [12][13]. This dissertation, on the contrary, attempts to decouple MLP modeling from detailed cache simulation, which allows the technique to be applied not

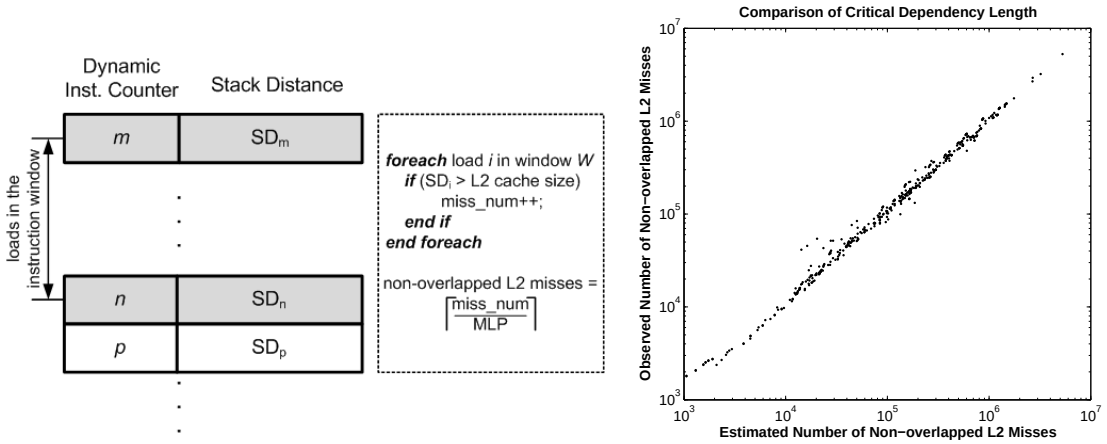
only in off-line resource demand analysis but also in on-line proactive performance prediction. To do so, the profiler generates the maximum number of loads LD_{max} in a dependency chain and the total number of loads LD_{total} in an instruction window. Then, LD_{total}/LD_{max} indicates the average number of loads that could be overlapped with each other in the instruction window. Assuming that the loads in a dependency chain have the same probability of missing L2 cache with other loads, LD_{total}/LD_{max} becomes the average number of the overlapped L2 load misses, or the MLP of the program. Meanwhile, the profiler also generates a load trace that contains the stack distance of a load and the dynamic instruction ID of the corresponding load, as shown in Figure 3.3(a). The MLP analyzer then walks through the trace, counts the number of L2 load misses that could happen in the instruction window for the given L2 cache size, and calculates the number of non-overlapped L2 misses by dividing the miss number with MLP. The total number of the non-overlapped L2 misses of the program is the sum of the non-overlapped misses in each instruction window:

$$N_{novp}(W, C) = \sum_i \left\lceil \frac{miss\_num(W, C)}{MLP} \right\rceil_i \tag{3.1}$$

where "$\lceil\ \rceil$" is the ceiling function, $miss\_num(W, C)$ is the number of L2 misses for the instruction window W and L2 cache size C. Figure 3.3(b) shows the accuracy of this model in estimating the number of non-overlapped L2 misses. The average error between the modeled number of non-overlapped misses and the simulated one is 9.3%, which is reasonably accurate for performance trend modeling.

3.2.5 Impact of Co-executing Threads

The basic performance model only captures the performance of a thread when it is executed alone on the processor core and is free to access all available intra-



(a) (b)

Figure 3.3: MLP modeling. (a) The estimation of non-overlapped L2 misses in the presence of MLP. (b) The accuracy of the estimated non-overlapped L2 misses. The results are based on the simulation of the 22 benchmark programs.

core resources. However, when multiple threads simultaneously execute on a core, these threads will compete each other for the shared intra-core resources, causing interference on the performance of each co-executing thread. In practice, to achieve controllable performance for each thread, the shared intra-core resources are dynamically partitioned among the threads [21] except for the issue/dispatch width, which often remains as shared such that one thread can exploit the full execution bandwidth when the other thread is waiting for its miss events to be served [32]. Under this circumstance, the effective issue width of each thread would be significantly different from the physical issue width, and the basic performance model needs to be augmented accordingly.

Assuming a processor with 2-way SMT and per-thread retirement capability, the effective execution rate of the thread can be estimated by analyzing the ILP of the co-executing threads. For example, if the ILP of thread T_0 (referred to as α_{T0})

and the ILP of thread T_1 (referred to as α_{T1}) are both larger than the issue width β of the processor core, on average each thread can execute at a rate equal to half of the issue width. If we could further obtain the fraction of the time that T_0 is serving a long-latency miss event, the effective execution rate of T_1 can be derived by considering the additional execution bandwidth T_1 has during that fraction of time. Similarly, if α_{T0} and α_{T1} are both smaller than β but the sum of these two is larger than β, on average the effective issue width of a thread is determined by the occupancy of its ready instructions: $\alpha_{T0} \cdot \beta/(\alpha_{T0}+\alpha_{T1})$ for T_0 and $\alpha_{T1} \cdot \beta/(\alpha_{T0}+\alpha_{T1})$ for T_1. By considering the fraction of the time in serving the long-latency miss events, the effective execution rate can be also derived. Table 3.1 summarizes the calculation of the effective execution rate under different scenarios. These values are used as the background steady-state execution rates of the performance model in the presence of SMT. Note that these estimations are based on the assumption that IQ uses the oldest-first policy to dispatch ready instructions.

Table 3.1: Estimation of Effective Average Execution Rate for 2-Way SMT

Cases:	**Effective Average Execution Rate**	
	Thread 0 (T_0)	Thread 1 (T_1)
$\alpha_{T0} < \beta, \alpha_{T1} < \beta,$ $\alpha_{T0} + \alpha_{T1} < \beta$	α_{T0}	α_{T1}
$\alpha_{T0} < \beta, \alpha_{T1} < \beta,$ $\alpha_{T0} + \alpha_{T1} > \beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \alpha_{T0} * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$
$\alpha_{T0} > \beta, \alpha_{T1} < \beta,$ $\alpha_{T0} + \alpha_{T1} < 2\beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$
$\alpha_{T0} > \beta, \alpha_{T1} < \beta,$ $\alpha_{T0} + \alpha_{T1} > 2\beta$	$\frac{2*\beta-\alpha_{T1}}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\alpha_{T1}}{2.0} * (1 - f_{T0}) + \alpha_{T1} * f_{T0}$
$\alpha_{T0} < \beta, \alpha_{T1} > \beta,$ $\alpha_{T0} + \alpha_{T1} < 2\beta$	$\frac{\alpha_{T0}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T1}) + \alpha_{T0} * f_{T1}$	$\frac{\alpha_{T1}*\beta}{\alpha_{T0}+\alpha_{T1}} * (1 - f_{T0}) + \beta * f_{T0}$
$\alpha_{T0} < \beta, \alpha_{T1} > \beta,$ $\alpha_{T0} + \alpha_{T1} > 2\beta$	$\frac{\alpha_{T0}}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{2*\beta-\alpha_{T0}}{2.0} * (1 - f_{T0}) + \beta * f_{T0}$
$\alpha_{T0} > \beta,\ \alpha_{T1} > \beta$	$\frac{\beta}{2.0} * (1 - f_{T1}) + \beta * f_{T1}$	$\frac{\beta}{2.0} * (1 - f_{T0}) + \beta * f_{T0}$

3.3 Summary

This chapter presents the details of the proposed analytical performance model that will be used in the following chapters. Compared with the existing performance models, the proposed model is decoupled from detailed cache simulations, yet is still able to capture the performance impact of different L2 cache sizes, a key capability that allows this model to be applied in efficient resource demand analysis and performance prediction. The proposed performance model is also able to capture the performance under different functional unit numbers, paving the way for proactive application scheduling in heterogeneous multi-cores with faulty FUs. Overall, the proposed performance model serves as the foundation for resource demand analysis, heterogeneity-aware application scheduling as well as dynamic resource management in SHMPs.

Chapter 4

Experiment Methodology

The previous chapter described the analytical performance model which is used as the common foundation of resource demand analysis, program-core mapping, application scheduling in static SHMP, and resource management in dynamic SHMP. Before going to the details of each of these techniques, this chapter summaries the simulation platforms, workloads, and metrics that are employed to evaluate the effectiveness and quality of these techniques.

4.1 Simulation Platform

4.1.1 Simulation Platform for Resource Demand Analysis

The simulation platform for resource demand analysis is built on top of SimProfile from Simplescalar tool set [33]. SimProfile is extensively modified to support the profiling of the statistics of the program characteristics needed by the performance model. The framework of resource demand analysis is evaluated on an out-of-order superscalar processor with two-level cache subsystem. The configuration ranges of relevant resources are listed in Table 4.1. Note that the associativity and the cache block size are kept constant across all possible L2 cache sizes as these aspects are not explored in this dissertation. The number of execution units is chosen such that the overall configuration is balanced. In total, the listed configurations cover over 100K design nodes. When evaluating the estimation of single-resource

demand, it is required that other resource configurations are fixed. However, due to the large design space, it is impossible for us to evaluate our framework exhaustively across all configurations. Therefore, this work uses three representative configuration sets: config-S(mall), config-M(edium), and config-L(arge), as the base configurations to evaluate our resource estimation model. The details of these configuration sets are also shown in Table 4.1.

Table 4.1: Configuration Options

Items	**Configuration Options**	**config-S**	**config-M**	**config-L**
Issue Width	1 :: 2x :: 8	1	4	8
ROB size	16 :: 2x :: 512	16	128	512
L2 D-Cache	64KB::2x::2048KB	64KB	512KB	2048KB
	8-way associative	8-way	8-way	8-way
	64B	64B	64B	64B
L1 I-cache	32KB	32KB	32KB	32KB
	2-way	2-way	2-way	2-way
	64B	64B	64B	64B
L1 D-cache	32KB	32KB	32KB	32KB
	4-way	4-way	4-way	4-way
	64B	64B	64B	64B
Branch Predictor(PAg)	1st-level: 8::2x::1K	1024	1024	1024
	2nd-level: 128::2x::4K	4096	4096	4096
Clock Freq.	0.5::0.1::2 (GHz)	0.5 GHz	1 GHz	2 GHz

For the evaluation of the resource demand analysis framework, this dissertation assumes the memory access latency to be 200ns, or 200 cycles at a clock frequency of 1 GHz. This latency number in terms of cycles scales proportionally with the operating frequency. The hit latencies of L1 and L2 caches are calibrated against Cacti 5.0 [34] under 90nm technology. The latencies of other execution units are also scaled to 90nm technology. The branch misprediction penalty is set to 20 cycles at 1 GHz. The power data of the interested processor configurations is collected using Wattch [35].

4.1.2 Simulation Platform for Program-core Mapping

The proposed framework for program-core mapping is evaluated on a hypothetical single-ISA heterogeneous quad-core processor. The detailed configurations of these cores are listed in Table 4.2. Each core is an out-of-order processor, and has a private 512K L2 cache with a hit latency of 12 cycles, and a miss latency of 200 cycles. Other parameters not shown in the table are chosen in a way that the design of the core is balanced. Since this work focuses on establishing the mapping relationship between programs and cores, the workloads are chosen to be independent with each other and the core-level communication is not considered in the evaluation of this part of the work. Note that the core-level communication (e.g., cache snooping) are symmetrical and tend to impose similar impact on the independent workloads, therefore, this simplifying assumption would not affect the validity of the evaluation.

Table 4.2: Configurations of Each Core

ITEMS	Configuration
Core 1	Out-of-order, single-issue, 1K Gshare, 32KB 4-way L1 d-cache 64byte, 4k 2-way i-cache 64byte, 512k L2 cache, L2 access latency 12 cycles
Core 2	Out-of-order, 2-issue, 2K Gshare, 64KB 4-way L1 d-cache 64byte, 8k 2-way i-cache 64byte, 512k L2 cache, L2 access latency 12 cycles
Core 3	Out-of-order, 4-issue, 4K Ghsare, 64 KB 4-way L1 d-cache 64byte, 16k 2-way i-cache 64byte, 512k L2 cache, L2 access latency 12 cycles
Core 4	Out-of-order, 8-issue, 8K Gshare, 128 KB 4-way L1 d-cache 64byte, 16k 2-way i-cache 64byte, 512k L2 cache, L2 access latency 12 cycles

Similar with the evaluation of resource demand analysis, the program-core mapping framework also uses SimProfile from Simplescalar tool set [33] to profile programs and collect the needed program characteristics, including instruction de-

pendence distance distribution, stack distance distribution. Again, Wattch [35] is employed to obtain the performance and power data for each benchmark program.

4.1.3 Simulation Platform for Application Scheduling in Static SHMP

The application scheduling in static SHMP is evaluated on a quad-core SPARCv9 OpenSolaris system modeled by a full system simulator Simics [36], extended with the GEMS toolset [37]. Each core in the CMP is 4-issue out-of-order processor modeled by Opal [37]. The simulated CMP system also contains a detailed memory subsystem model, which includes an inter-core last-level cache network and a detailed memory controller. In addition, the simulated system supports software data prefetching. Table 4.3 lists the nominal configurations of the CMP system in detail. The dynamic power of the processor cores is estimated by Wattch [35], and the leakage power on caches and other SRAM structures in the core is estimated by Cacti 5 [34]. The simulator also uses Orion [38] to estimate the power on the interconnection network of the last-level cache. Therefore, the performance and power overhead of application migration is fully modeled.

This work focuses on the core-level heterogeneity on frequency, Integer ALU (I-ALU) number and L2 cache size, yet it is infeasible to evaluate every possible configuration. Therefore, this dissertation evaluates three sets of heterogeneous configurations created by varying these resources over their nominal values, as shown in Table 4.4. These configuration sets are organized as low heterogeneity (LH) where only frequency varies, medium heterogeneity (MH) where both frequency and I-ALU number vary, and high heterogeneity (HH) where all three resources vary. While there are other heterogeneous configurations, these three configuration sets cover the representative degrees of heterogeneity caused by manufacturing imperfection.

Table 4.3: Nominal Configurations of the CMP system

	Parameter	Configurations
Core	Clock Frequency	4GHz
	Fetch/Issue/Commit Width	4/4/4
	Ld/St Units	2/2
	I-ALU/FP Units/FP Multipliers	4/2/2 (fused multiply/add for I-ALU)
	ROB size	128
	Load/Store Queue Size	32/32
	Branch Predictor	YAGS, 16 PHT bits, 10 Tag bits
Cache	L1 I-Cache/D-Cache	32KB, 2-way, 64Byte, LRU, 1-cycle
	L2 Cache	2MB per core, 8-way, 64Byte, LRU, 12-cycle
	L2 MSHR Entry	32
	Coherence Protocol	Directory-based MOESI
Memory	Size/Model	4GB/DDR2-800
	Controller	FR-FCFS policy [39]
	Organization	8 banks per rank, 2 ranks per DIMM

Table 4.4: Configurations of the CMP system

Parameter	**Configurations**											
	Low Heter.				**Medium Heter.**				**High Heter.**			
	C-0	C-1	C-2	C-3	C-0	C-1	C-2	C-3	C-0	C-1	C-2	C-3
Freq.(GHz)	4	3.6	3.2	2.8	4	3.6	3.2	2.8	4	3.6	3.2	2.8
I-ALU	4	4	4	4	4	2	3	1	4	2	3	1
L2 Cache (MB,Ways)	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	2, 8	1.5, 6	1, 4	0.5, 2

4.1.4 Simulation Platform for Resource Management in Dynamic SHMP

The resource management in dynamic SHMP is also evaluated on a quad-core SPARCv9 OpenSolaris system modeled by a full system simulator Simics [36], extended with the GEMS toolset [37]. Each core in the CMP is a 4-issue out-of-order processor and supports 2-way SMT with ICOUNT [23] fetch policy. The simulated CMP system also contains a detailed memory subsystem model, which includes an inter-core last-level cache network and a detailed memory controller. Table 4.5 lists the configurations of the CMP system in detail. Again, the dynamic power of the processor cores is estimated by Wattch [35], and the leakage power on caches and other SRAM structures in the core is estimated by Cacti 5 [34]. Orion [38] is used

to estimate the power on the interconnection network of last level caches.

The ROB size is partitioned at the granularity of 32 entries. Other intra-core resources such as issue queue size and physical register number are partitioned in proportion to the ROB size. Each thread is guaranteed to have at least 32 entries of ROB size. The L2 cache is shared among the threads, and its size is partitioned at the granularity of cache ways, with each thread allocated with at least one cache way. The CMP system supports per-core DVFS, with the frequency of each core ranging from 2GHz to 4GHz at the step of 0.1GHz. This dissertation assumes that the CMP system reaches the power budget when it is fully loaded and each core is running at 3GHz.

Table 4.5: Configurations of the CMP system

	Parameter	**Configurations**
Core	Maximum Clock Frequency	4GHz
	Fetch/Issue/Commit Width	4/4/4
	Ld/St Units	2/2
	I-ALU/FP Units/FP Multipliers	4/4/2 (fused multiply/add for I-ALU)
	ROB size/Issue Queue	256/160
	Load/Store Queue Size	64/64
	Branch Predictor	YAGS, 16 PHT bits, 10 Tag bits
	Physical Register Number	380
Cache	L1 I-Cache/D-Cache	32KB, 2-way, 64Byte, LRU, 1-cycle
	L2 Cache	16MB, 32-way, 64Byte, LRU, 12-cycle
	L2 MSHR Entry	32
	Coherence Protocol	Directory-based MOESI
Memory	Size/Model	4GB/DDR2-800
	Controller	PAR-BS policy [40]
	Organization	8 banks per rank, 2 ranks per DIMM

4.2 Workloads

4.2.1 Workloads for Program Resource Demand Analysis

The workload to evaluate the framework of resource demand analysis is composed of 22 SPEC CPU2006 programs [41] (*gamess*, *dealII*, *calculix*, *povray*, *tonto*, *lbm*,*wrf* are not included in the workload as they fail to compile to Alpha ISA). Each of the 22 programs is compiled to Alpha-ISA with peak configurations. To speedup the evaluation, the single Simpoint interval with 100 million instructions [42] is used to represent each program.

4.2.2 Workloads for Program-core Mapping

The workload of program-core mapping is composed of programs from SPEC CPU2000, SPEC CPU2006 and MediaBench. Each program is compiled to Alpha-ISA configurations. This work uses the single Simpoint interval with 100 million instructions [42] for each SPEC CPU program. Programs from the MediaBench suite are chosen in such a way that the program's dynamic instruction count is comparable with those Simpoint intervals for the SPEC CPU2000 suite.

4.2.3 Workloads for Application Scheduling in Static SHMP

To stress the application scheduling in heterogeneous CMPs, the workload also needs to be heterogeneous because homogeneous workloads, such as the threads from the same multi-threaded program, benefit little from swapping tasks in heterogeneous CMPs [3]. Therefore, this dissertation uses multi-programmed workloads composed of the programs from the SPEC CPU2006 benchmark suite [41], with each compiled to SPARC ISA. Specifically, this dissertation constructs 9 heterogeneous workloads, each containing 2 integer programs and 2 FP programs, as shown

in Table 4.6. The program mix is based on the similarity analysis by Phansalkar et al. [43], and is created such that: a) the workloads cover all representative benchmark; b) programs in each workload are from clusters with large linkage distance [43]. Each workload would be running on the aforementioned three heterogeneous CMP systems. Due to the limitation of the simulator, it is extremely difficult to synchronize the programs in the workload at their own simulation points. Therefore, in this work, each workload is fast-forwarded for 3 billion instructions to reach its steady state execution, and then uses the next 100 million instructions to warmup the cache subsystem. Each work runs on the simulated system for a time span equivalent to 0.2 seconds on a real 4GHz CMP system, which covers up to 1 billion instructions for a program. The scheduling interval is set to 10ms, which is standard in *OpenSolaris*. Therefore, each simulation provides 20 scheduling epochs.

Table 4.6: Workloads and Their Characteristics

Program Mix	**Symbol**	**Category (Memory/CPU)**
mcf,bwaves,povray,gcc	mbpg	Mem-Mem-CPU-CPU
xalancbmk,namd,lbm,omnetpp	xnlo	CPU-CPU-Mem-Mem
libquantum,xalancbmk,wrf, soplex	lxws	Mem-CPU-Mem-Mem
milc, soplex, omnetpp, sjeng	msos	Mem-Mem-CPU-CPU
leslie3d,sphinx3,hmmer,astar	lsha	Mem-CPU-CPU-CPU
zeusmp, libquantum, omnetpp, tonto	zlot	Mem-Mem-CPU-CPU
calculix, dealII, perlbench, bzip2	cdpb	CPU-CPU-CPU-CPU
povray, mcf, cactusADM, astar	pmca	CPU-Mem-Mem-CPU
milc, gobmk, lbm, gcc	mglg	Mem-CPU-Mem-CPU

4.2.4 Workloads for Resource Management in Dynamic SHMP

The workload to evaluate the resource management in dynamic SHMP is composed of the programs from the SPEC CPU2006 benchmark suite [41], with each compiled to SPARC ISA. This dissertation uses 12 heterogeneous multiprogrammed

workloads, each containing 8 programs, as shown in Table 4.7. These workloads are grouped into three categories: CPU-intensive (high-ILP), memory-intensive, and the mixture of both. Each workload will be running on the aforementioned CMP systems. Again, due to the limitation of the simulator, simulation point is not employed. Instead, for each run, each workload is fast-forwarded for 4 billion instructions to reach its steady state execution, and then uses the next 100 million instructions to warmup the cache subsystem. Then, each workload runs on the simulated system for 200M instructions to evaluate the performance of various resource allocation policies.

Table 4.7: Workloads and Their Characteristics

Workload Mix	**Symbol**	**Category**
povray, calculix, sjeng, hmmer perlbench, wrf, dealII, tonto	pcshpwdt	ILP
gcc, povray, astar, calculix gobmk, hmmer, bzip2, dealII	gpacghbd	
astar, bzip2, gobmk, povray sjeng, perlbench, dealII, gamess	abgpspdg	
namd, gcc, gromacs, perlbench h264ref, tonto, sphinx3, sjeng	nggphtss	
mcf,omnetpp,bwaves,lbm povray, namd, gcc, xalancbmk	moblpngx	MIX
dealII, sjeng, libquantum, omnetpp povray, soplex, perlbench, milc	dslopspm	
libquantum, cactusADM, xalancbmk calculix,wrf,mcf, soplex, omnetpp	lcxcwmso	
leslie3d,tonto,sphinx3, omnetpp hmmer, libquanutm, astar, zeusmp	ltsohlaz	
soplex, xalancbmk, milc, lbm mcf, cactusADM, zeusmp, leslie3d	sxmlmczl	MEM
leslie3d,soplex, zeusmp, bwaves wrf, cactusADM, xalancbmk, lbm	lszbwcxl	
lbm, milc, xalancbmk, leslie3d zeusmp, wrf, mcf, soplex	lmxlzwms	
milc, xalancbmk, mcf, cactusADM soplex, leslie3d, bwaves, wrf	mxmcslbw	

4.3 Metrics

The metrics to evaluate program-core mapping are Energy-Delay-Product (EDP) and Makespan. Makespan is the time between the start and finish of a group of programs, and is considered to the appropriate metric for throughput of a batch of programs [44]. For the application scheduling and resource management, this dissertation uses the aggregated throughput, defined as the sum of each application's million-instructions per second (MIPS) to evaluate the system performance. To enforce fairness, this dissertation also evaluates the performance using the weighted speedup [45], which is defined as $\sum_i IPC_i^{scheduled}/IPC_i^{ref}$. To measure the system efficiency, this dissertation uses the metric $mips^3/W$, which is inverse to energy-delay-square (ED^2) and has been accepted as the efficiency metric for high-performance systems [46].

Chapter 5

Program Resource Demand Analysis

This chapter presents the framework for *Program REsource Demand Analysis* (PREDA), which uses the performance model described in Chapter 3 as the key component to evaluate the performance under a specific resource allocation. The evaluation of the estimated resource demands is also presented in this chapter.

5.1 Resource Demand Definition

Before proceeding to the details, it is important to make a clear definition of resource demand. The meaning of resource demand contains two elements: the target performance and the energy efficiency. On one hand, different target performance levels may lead to different resource requirements. Specifically, as the target performance level increases, the amount of resource required may also increase in order to meet the performance target. On the other hand, for a given performance target, there may be a set of different amounts of resources being able to meet that target. Among them, only the one that is energy efficient is chosen. Therefore, this dissertation introduces the following resource demand definition:

***Definition**: Resource Demand $D(p)$ is the amount of resource a thread requires to efficiently achieve no less than $p\%$ of the maximum performance achieved with the entire resources allocated to the thread.*

This definition assumes performance monotonicity, which means the perfor-

mance of a thread increases monotonically as the amount of resource allocated to the thread increases [47]. Note also that this definition uses relative performance instead of absolute performance as the performance target. This is because the absolute performance targets, such as the Instruction-Per-Cycle (IPC) rate, may lead to *ill-defined* cases where the target cannot be satisfied no matter how many resources are allocated. The relative performance target avoids this problem, and more importantly it is inline with the satisfiability of the QoS target proposed by Guo, *et al.* [7]. In fact, with this definition, our framework can be treated as a conversion layer that converts the performance targets into the resource demands, which could be used as the *Resource Usage Metrics* for QoS enforcement [7].

5.2 Overview of the Framework

The proposed PREDA framework consists of two parts: the program characteristics profiler and the PREDA kernel. As shown in Figure 5.1, the program profiler walks through the dynamic instruction stream and extracts a set of program characteristics, which contain instruction dependency chain distribution, stack distance distribution, instruction mix, and branch transition rate and its access frequency. These characteristics are then fed to the PREDA kernel, which consists of an ILP model, an MLP model, a performance model and a resource analyzer. The models for ILP and MLP are responsible for translating the program characteristics into the ILP and MLP information that can be directly used by the performance model. Hence, these models serve as the key layer to decouple the performance evaluation from detailed simulations. The performance model takes the ILP and MLP information along with the branch predictability characteristics and estimates the program execution time on an out-of-order processor. The resource analyzer

converts the estimated performance into the relative performance, and searches the configuration space for the amount of resources required to meet the performance targets. The estimated resource demands include processor issue width, processor reorder buffer (ROB) size, L2 (or last level) cache sizes, operating frequency, memory bandwidth and branch predictor size. These resource demands are estimated either in single-resource mode (other resources are fixed) or in multi-resource mode (combinations of changing resources). Note that the proposed performance model could also be applied online by using a set of on-line profilers such as stack histogram profiler [25][26] and critical path predictor [48]. However, this dissertation focuses on evaluating the accuracy and the complexity of off-line demand estimation.

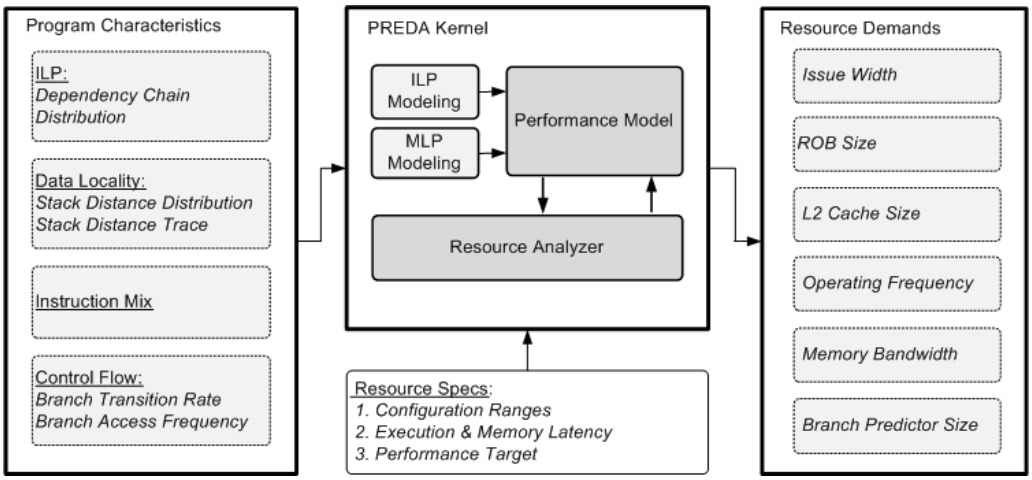


Figure 5.1: The PREDA framework.

5.3 Performance Modeling

The performance model used in this framework is described in Chapter 3. Note that since this framework addresses the resource demand estimation for single-threaded execution, the performance model does not need to model the impact of co-executing threads.

5.4 Demand on Multiple Resources

In this dissertation, the estimation of multi-resource demands is built on top of the single-resource demand estimation, which uses the marginal utility to determine the demand on the corresponding resource. The marginal utility originates from economic theory, and is defined as the ratio between the incremental utility over the amount of incremental resource. It has been successfully used as the metric for last-level cache partitioning [25][26]. This dissertation further extends the application of marginal utility to different hardware resources, and defines the marginal utility as follows:

$$MarginalUtility(D_\beta) = \frac{Perf(RES_\beta + D_\beta) - Perf(RES_\beta)}{D_\beta} \tag{5.1}$$

where RES_β is the amount of resource β, and D_β is the amount of increment in resource β. Note that the maximum marginal utility represents the best (or most efficient) use of a resource increment. Therefore, with marginal utility, the problem of resource demand estimation is transferred to the problem of finding the amount of resource that meets the performance target meanwhile has the maximum marginal utility. Thus, the estimation of the single resource demand becomes straightforward: sweeping the interested resource from its minimum to its maximum while keeping other resources fixed, and searching for the amount of resource that satisfies the performance target and has the largest marginal utility. However, there is an exception: when the performance with the minimum resource allocation is larger than the target performance, the resource demand is set to the minimum value.

While the single-resource demand estimation is straightforward, the estimation of multi-resource demands is non-trivial because the marginal utility is only comparable among the resources with the same type. To address this problem,

Pseudocode 1 Demand on Multiple Resources

```
#define N
/*the number of resources that could change simultaneously*/
#define max_resource_array[N]
/*the array of maximum available resources*/
#define eval_perf(resource_array)
/*Evaluate the execution time with the resource configuration array resource_array*/
#define est_demand(resource_array, i, target_perf)
/*Estimate the demand of resource i under the performance target target_perf*/

for ( i=0; i < N; i++)
    base_demand[i] = est_demand(max_resource_array,i,target_perf);
    /* estimate the demand for resource i when other resources are set to maximum*/
end for
while( TRUE )
    perf = eval_perf(base_demand);
    if( perf > target_perf )
          set the final demands as the base demand estimates;
          break;
    else
          for ( i=0; i < N; i++)
             temp_demand[0..N] = base_demand[0..N];
             /* copy the base resource demand to temp_demand array */
             new_demand[i] = est_demand(base_demand,i,target_perf);
             temp_demand[i] = new_demand[i];
             perf_gain[i] = perf - eval_perf(temp_demand);
             /* calculate the performance gain with the newly */
             /* estimated resource demand */
          end for
    find the index max_index of the maximum value in array perf_gain[N];
    base_demand[max_index] = new_demand[max_index];
end while
```

this dissertation proposes an algorithm based on the gradient performance gain, as shown in Pseudocode 1. The first step of this algorithm is to estimate the demand on each resource individually when other resources are configured to be the maximum. The estimated single-resource demands are then combined together as the

initial multi-resource configuration, which serves as the starting point of the iterative searching process. Each iteration estimates the single-resource demand based on the previously estimated multi-resource configuration, and calculates the corresponding performance gain over the performance of the multi-resource configuration estimated in the previous iteration. The resource with the maximum performance gain is selected to update the multi-resource configuration, and the process continues until the performance meets the target. The complexity of this algorithm is $O(n \cdot k)$, where k is the number of iterations, and n is the number of the changing resources. This algorithm can estimate the multi-resource demands on four types of resources, including ROB size, issue width, L2 cache size, and frequency.

5.5 Demand on Memory Bandwidth

The program's memory bandwidth requirement is important for CMP systems, where multiple programs share the limited memory bandwidth resource. It is composed of the requirements for memory read bandwidth and memory write bandwidth. Assuming a write-back L2 cache, a read request to the main memory can be triggered by a load/store miss, and a write request can only occur when a dirty cache block is evicted (i.e., cache write-back). While the conventional stack distance model can capture the read traffic to the memory, it is unable to estimate the write-back traffic. To solve this problem, the conventional stack distance model needs to be augmented to capture both reads and write-backs to the main memory.

To do so, during stack distance profiling, each cache block is associated with a dirty bit and mark the dirty bit whenever the block has been written to. Then, a *Dirty Stack Histogram* is used to record the largest stack distance of a dirty cache

Pseudocode 2 Update of the Dirty Stack Histogram

```
if( dirty == 1 )
    if( the block was most recently accessed by a read
    && stack_distance > dirty_stack_distance)
            dirty_histogram[dirty_stack_distance]- -;
            dirty_histogram[stack_distance]++;
            dirty_stack_distance = stack_distance;
    else if( the block was most recently accessed by a write)
            dirty_histogram[stack_distance]++;
            dirty_stack_distance = stack_distance; end if
end if
if( the current access is a write )
    dirty = 1;
    dirty_stack_distance = 0;
end if
```

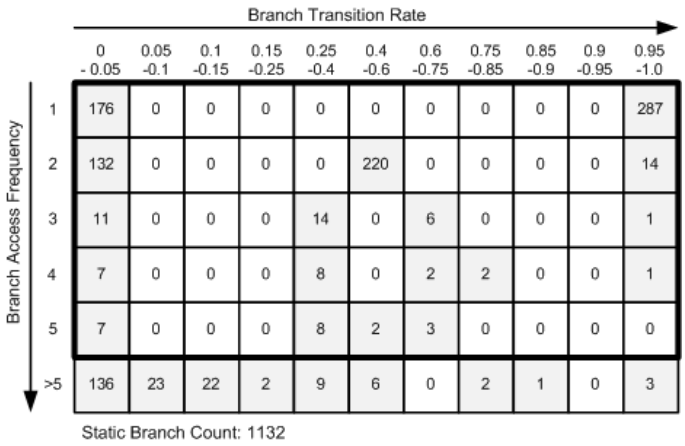
block. The reason for only considering the largest stack distance is to avoid multiple write-back counts for one store. The details of updating the dirty stack histogram are described in Pseudocode 2. Note that once the dirty bit is set, it will never be reset during profiling. Therefore, the dirty bit is unaware of multiple writes to the same block at different stack distances, which may lead to multiple write-backs under certain cache sizes. To handle this situation, the dirty block needs to be differentiated according to whether the block was most recently accessed by a read or a write. Specifically, if the dirty block was most recently accessed by a write, the corresponding counter in the dirty histogram will be incremented regardless of the stack distance. With the dirty histogram, one is able to estimate the number of dirty evictions by using the property of the conventional stack distance model. Specifically, a dirty eviction happens whenever the dirty stack distance of a block is larger than the given cache size.

5.6 Demand on Branch Predictor Size

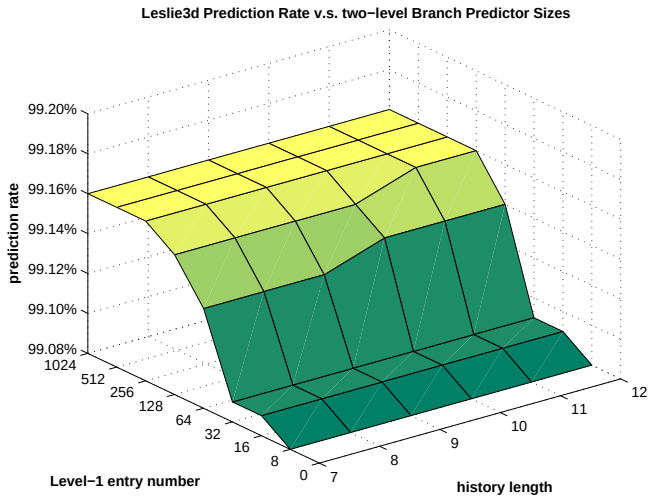
The purpose of estimating the program's demand on branch predictor size is to prevent unnecessary resource over-provisioning for the branch predictor. However, due to the lack of analytical models that translate the predictor size to the prediction accuracy, the demand of branch predictor size may have to be estimated *directly* based on the program's branch characteristics. Moreover, since different types of predictors may yield different prediction accuracy levels, the demand on branch predictor size has to be estimated in an *ad hoc* way. The current implementation of PREDA only supports estimating the demand on the two-level PAg predictor size [49].

PREDA estimates the demand on predictor size based on two branch characteristics: the branch transition rate, and the branch access frequency. As mentioned previously, branch transition rate has an implication on branch history length. Branches with very high or very low transition rate are easy to predict and only require short history registers; whereas branches with near 50% transition rate are hard to predict and require long history registers. However, branch transition alone could not tell how often a branch is executed in the dynamic instruction stream. For those branch instructions with very few accesses, they have negligible effect on the overall IPC whether they are predicted correctly or incorrectly. Therefore, these branches should be filtered when determining the demand of branch predictor size. Note that these two branch characteristics are in correspondence with the two-level PAg branch predictor, where the first level table (Per-Address History Table) is essentially a cache holding the frequently accessed branches, and the second level is indexed with a history register reflecting the predictability of the branches. As

an example, Figure 5.2(a) shows the branch transition rate distribution as well as branch access frequency distribution of the SPEC CPU program *leslie3d*. The total static branch count is 1132, which seems to indicate that the first-level table should contain 1K entries. However, if the branch instructions with small access frequencies (less than 5 in this case) are filtered out, the static branch count becomes 204,



(a)



(b)

Figure 5.2: Branch predictor size demand estimation

indicating that 256 entries in the first-level table would be sufficient. This is proved by Figure 5.2(b), which shows that the prediction accuracy does not degrade until the first-level entry is smaller than 256.

Pseudocode 3 Demand on Branch Predictor Size

```
#define access_threshold   16
while( TRUE )
   foreach static branches
         if ( branch_access_frequency < access_threshold )
            filtered_static_branch ++;
            filtered_dynamic_branch=filtered_dynamic_branch+branch_access_frequency;
            end if
   end foreach
   if (filtered_dynamic_branch < 0.001*total_dynamic_branch) break;
   else access_threshold - -; end if
end while
 first_level_entry = total_static_branch - filtered_static_branch;
foreach remaining branches
   if ∃transition_rate∈ [0.4, 0.6]
         history_length= max_history;
         /* max_history is the maximum history length
         specified in the design space */
   else if ∃transition_rate∈ [0.25, 0.4) ⋃ (0.6, 0.75]
         history_length=(max_history-1)>min_history ? max_history-1:min_history;
         /* min_history is the minimum history length
         specified in the design space */
   else if ∃transition_rate∈ [0.15, 0.25) ⋃ (0.75, 0.85]
         history_length=(max_history-2)>min_history ? max_history-2:min_history;
   else if ∃transition_rate∈ [0.1, 0.15) ⋃ (0.85, 0.9]
         history_length=(max_history-3)>min_history ? max_history-3:min_history;
   else if ∃transition_rate∈ [0.05, 0.1) ⋃ (0.9, 0.95]
         history_length=(max_history-4)>min_history ? max_history-4:min_history;
   else if ∃transition_rate∈ [0, 0.05) ⋃ (0.95, 1.0]
         history_length=(max_history-5)>min_history ? max_history-5:min_history;
   end if
end foreach
```

Based on this observation, this dissertation proposes the heuristics shown in

Pseudocode 3 to estimate the demand on the first level table size and the branch history length. Note that in order to prevent branch filtering from aggressively impacting the prediction accuracy, this heuristic ensures that the total number of filtered dynamic branches is less than 0.1% of the total dynamic branches. Note also that the transition rate buckets used for determining the history length are consistent with those used in branch classification by Haungs, *et al.* [50].

5.7 Evaluation

The evaluation of the proposed framework covers three major aspects: the accuracy of the models, the accuracy of resource demand estimation and the computation complexity of the framework. The details of the experiment platform is shown in Chapter 4.

5.7.1 Model Accuracy

Since the resource demand estimation is based on the relative performance as opposed to the absolute one, it is necessary to validate whether the performance model could accurately capture the performance trend as the resource allocation changes. To do so, we sweep the resource allocations and calculate the corresponding throughput with the performance model, and then normalize them with respect to the largest throughput. The normalized throughput curve is compared against the one obtained from detailed simulation. Figure 5.3 shows an example of such comparison for *bzip2*. Ideally, these two curves should be overlapped with each other. However, due to the imperfection of the performance model, the estimated performance curve deviates from the simulated one. To measure the difference between these two curves, the absolute difference of the normalized throughput is calculated

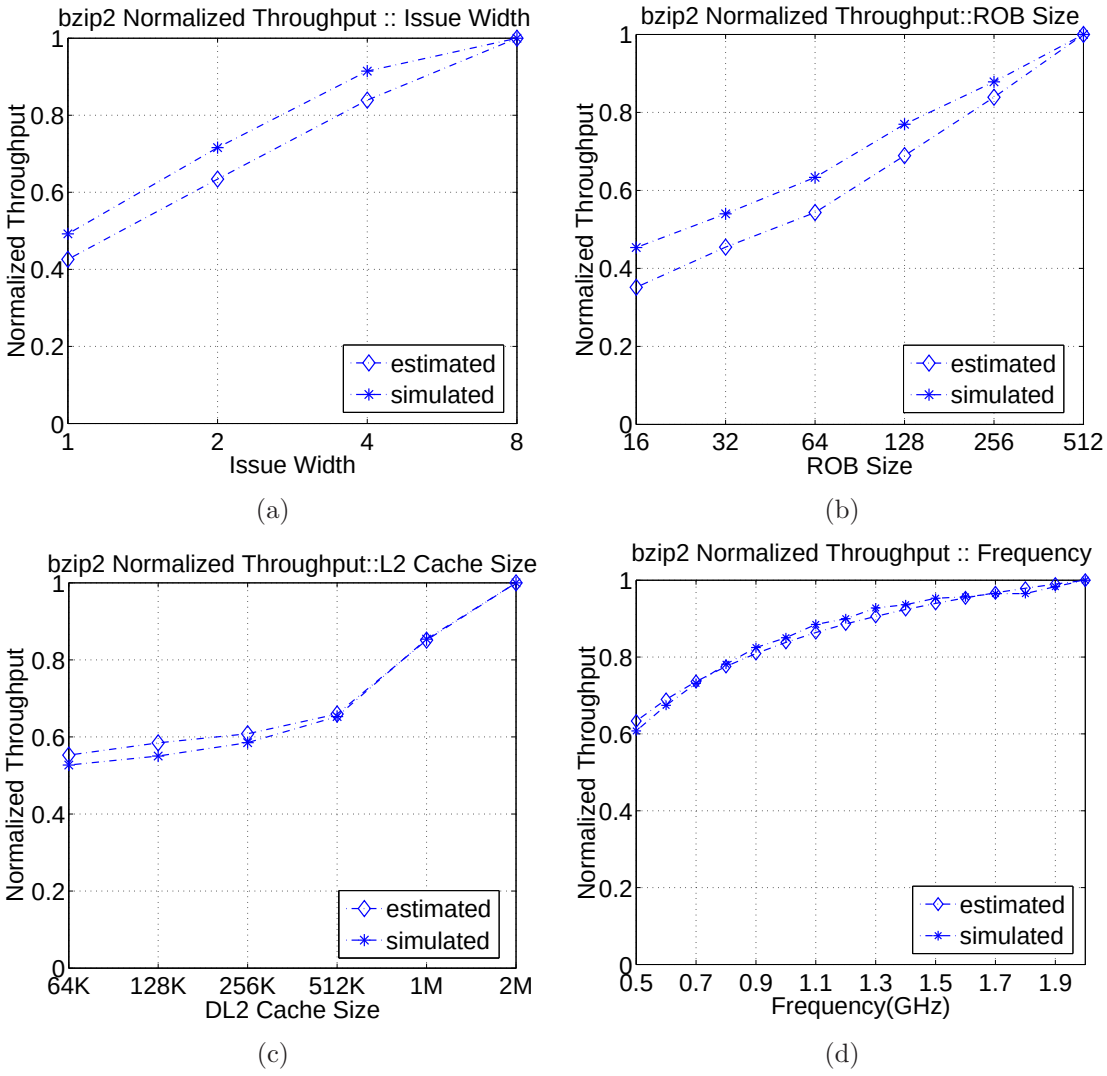


Figure 5.3: The comparison of normalized throughput for *bzip2* as one of the resources changes. The configurations of other unchanged resources follow config-M.

on each node of the curves, and then the average difference for each curve is also calculated to evaluate the accuracy of this model. Figure 5.4 summarizes these differences for each program. Note that most of the programs have a relatively large error in Config-L. This is mainly because some of the second-order effects, such as

the branch misprediction caused by speculative path information, becomes more significant in very wide machines; whereas our model only captures first-order effects. However, even in the worst case, the modeled performance trend is only on average 0.107 off the simulated one, which is still reasonable for resource demand estimation.

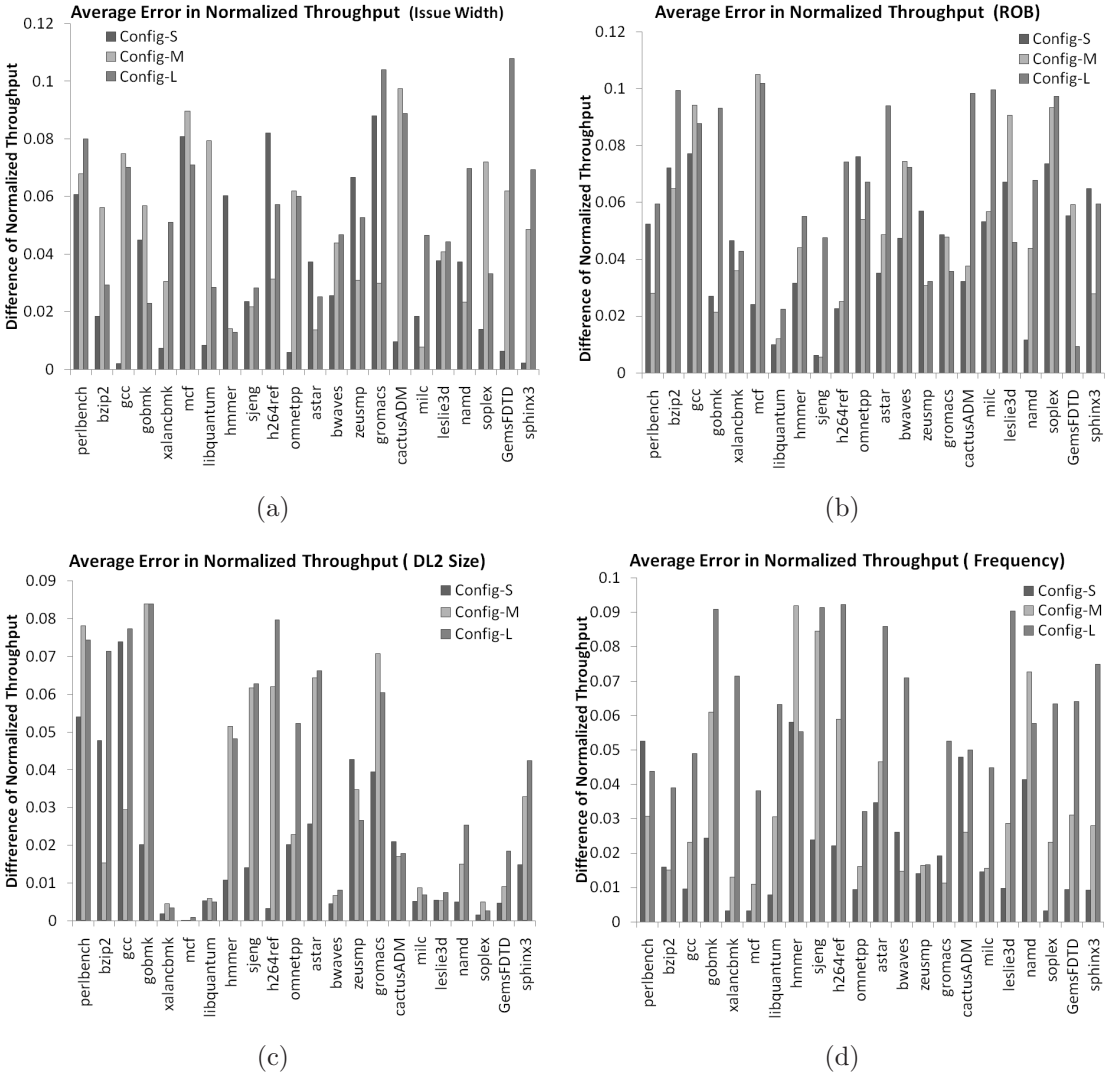


Figure 5.4: Average error of the normalized throughput for issue width, ROB size, L2 cache size, and frequency. Each resource estimation was evaluated on three configurations: config-S, config-M, and config-L

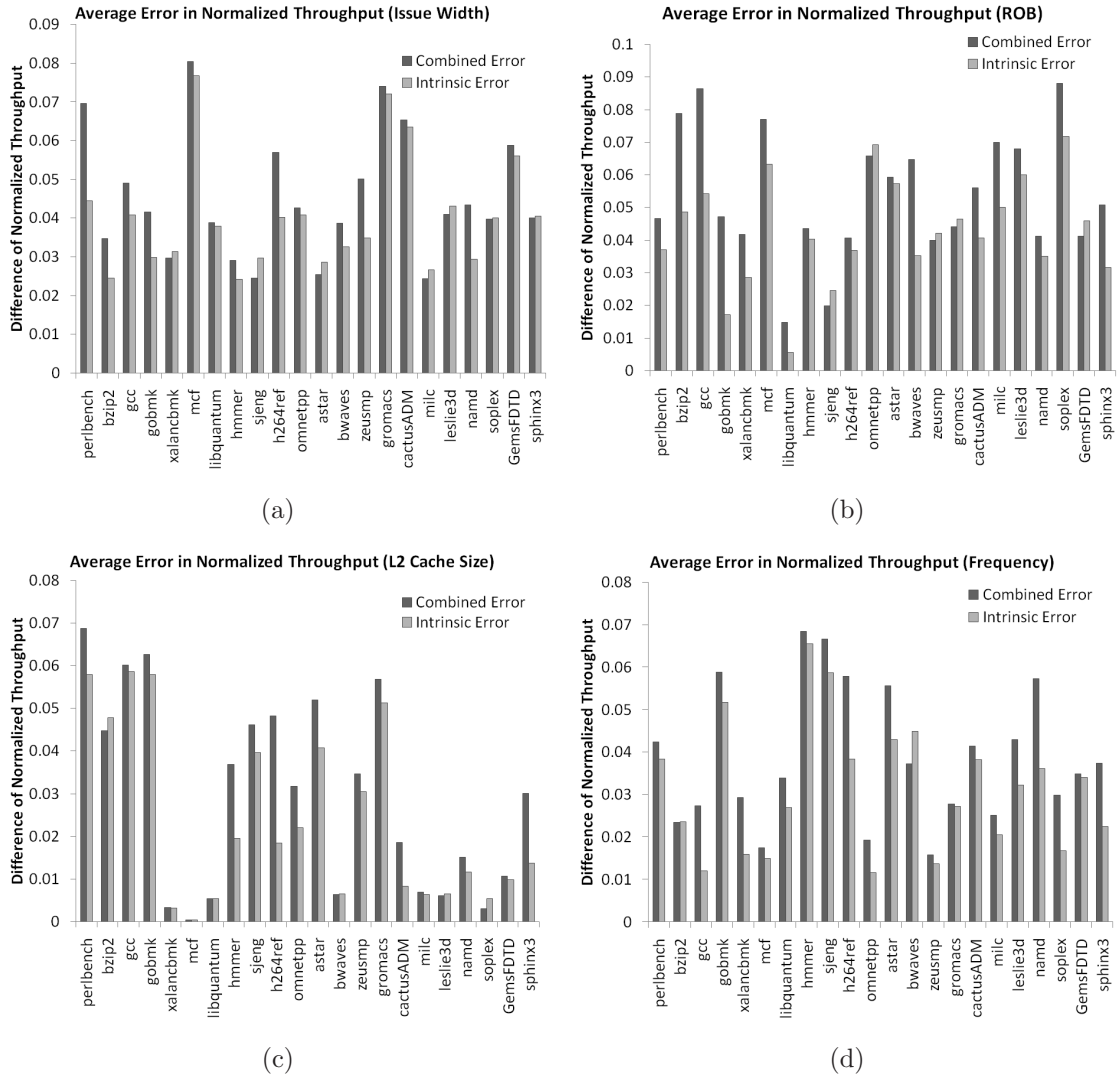


Figure 5.5: Comparison of the combined error and the intrinsic error in normalized throughput.The intrinsic error is obtained by using the simulated values of the non-overlapped L2 misses and the branch mispredictions in the performance model. The errors are averaged across three configurations: config-S, config-M, config-L.

The error of the performance model consist of two parts: the intrinsic error, which is the inherent modeling error caused by some simplifying assumptions of the model, and the parameter error, which is the error introduced by the estimation of

model parameters using program characteristics. Figure 5.5 shows the comparison between these two errors. As expected, most programs have much smaller intrinsic error than the combined one, especially for *gcc* and *namd*. However, some programs see a slightly higher intrinsic error than the combined error. This is because the parameter error and the intrinsic error may be canceling each other, leading to a smaller combined error. In worst case, the average intrinsic error is 7.6% in terms of the normalized throughput (*mcf* in Figure 5.5(a)), which shows the potential of applying the performance model on-line.

5.7.2 Accuracy of Resource Demand Estimation

5.7.2.1 Single-Resource Demand Estimation

This section evaluates the estimation of single-resource demand on issue width, ROB size, L2 cache size, and frequency at 20 different performance target levels, ranging from 0 to 95% with a step of 5%. Figure 5.6 shows the comparison between the demand estimated with our performance model and the one obtained from detailed simulation for program *bzip2*. Because of the imperfection in performance modeling, there are differences between the estimated and the simulated demands at certain performance targets. The average amount of these differences across the entire 20 performance target levels reflects the accuracy of the demand estimation, as shown in Figure 5.7. It can be observed that the demand difference at any performance target level is no larger than 4 configuration units. The largest demand difference happens in estimating the frequency demand, and this difference is still reasonable considering there are 16 different configuration options for frequency demand.

To evaluate the estimation of memory bandwidth demand, this dissertation

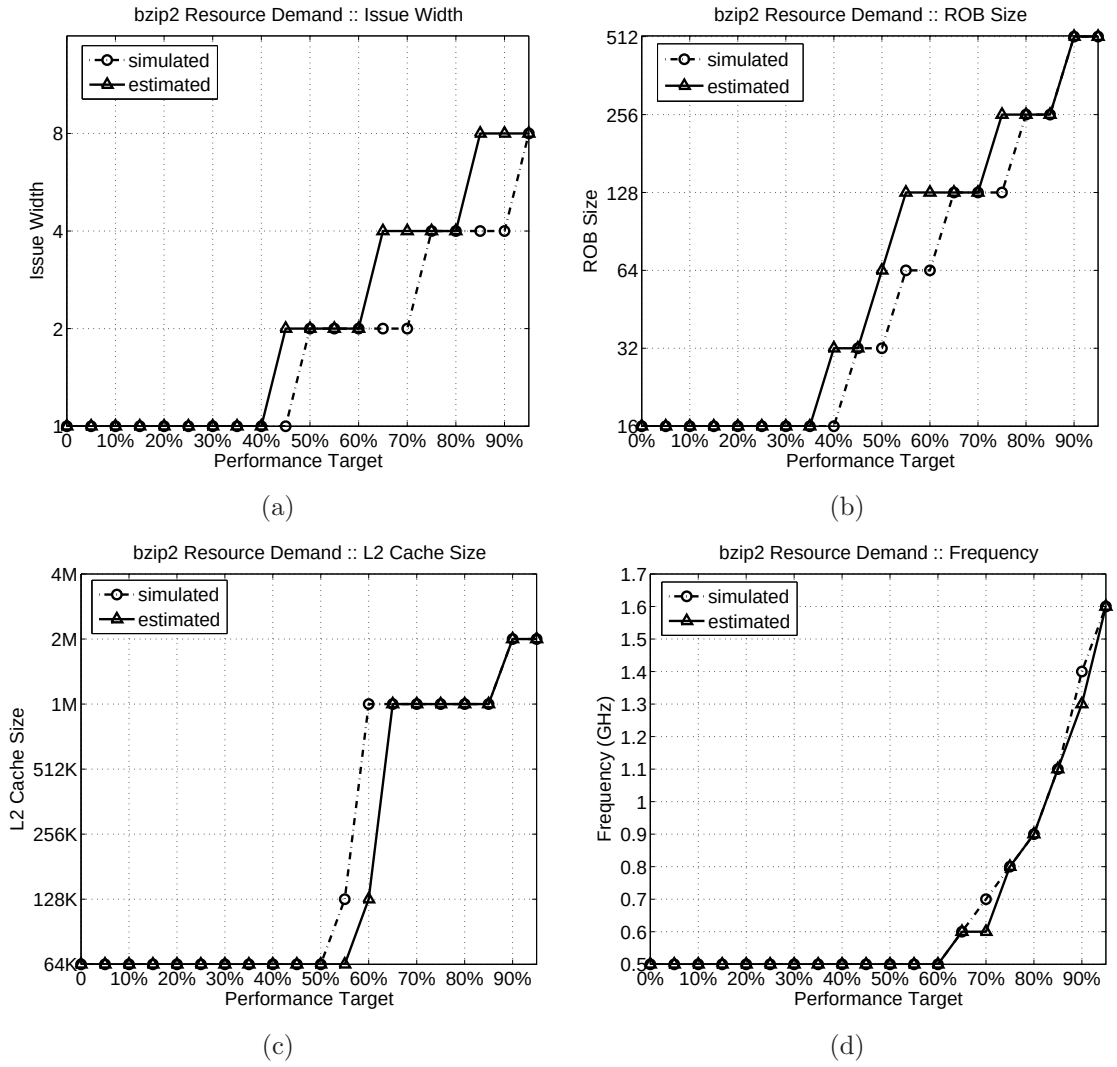


Figure 5.6: The accuracy of single-resource demand estimation for *bzip2*. The results are based on config-M.

compares estimated memory bandwidth with the simulated one at each 100K instruction interval, and accumulate the absolute difference between these two to obtain the overall memory bandwidth estimation error. Figure 5.8 shows the error rates of bandwidth demand estimation for both memory read and memory write traffics at three different configurations. On average, the total memory bandwidth estimation error

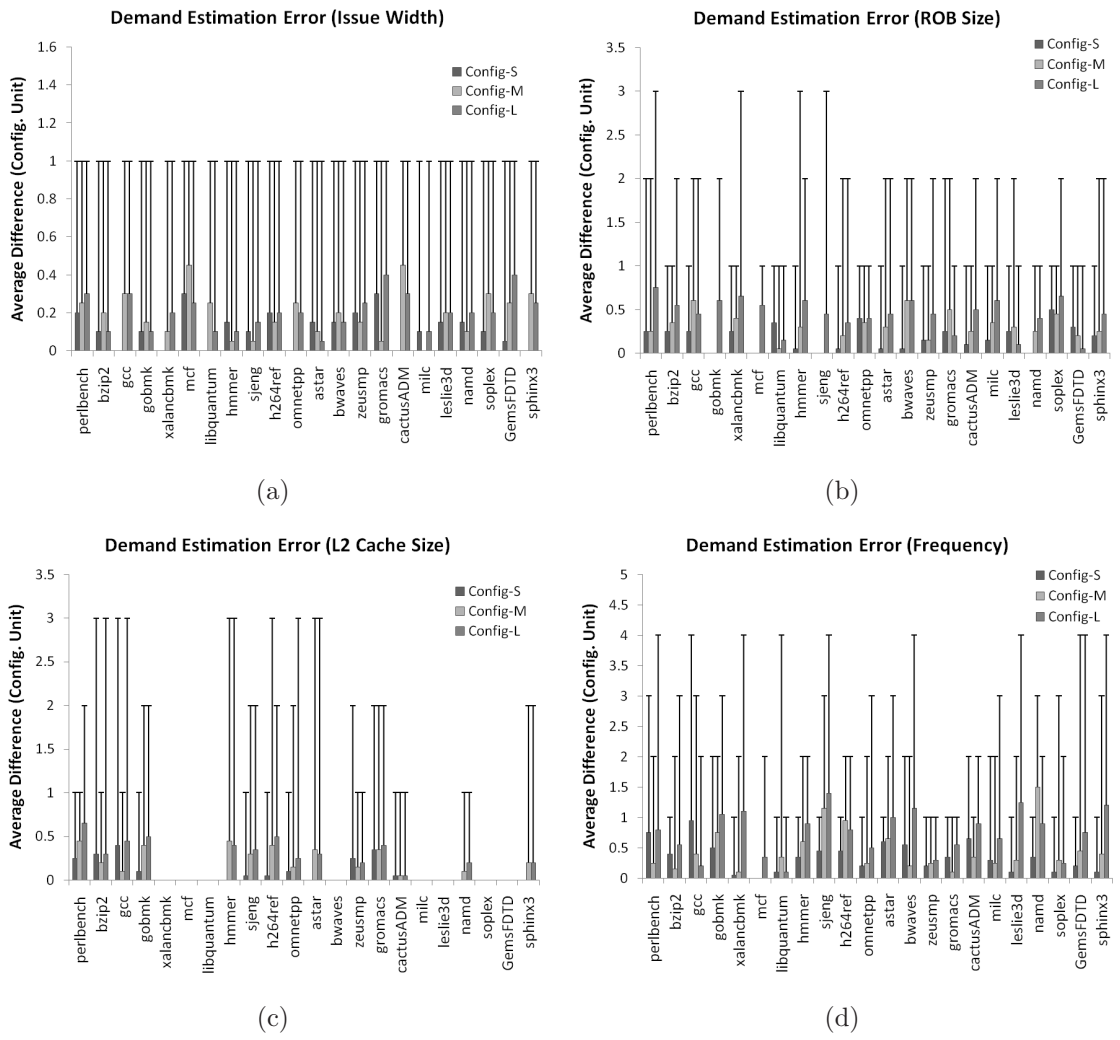


Figure 5.7: The error of resource estimation. Config unit refers to the quantization of each resource shown in Table 1. The error bar represents the largest error in demand estimation for the corresponding program.

increases from 4.76% to 6.26% as the L2 cache size changes from 64KB to 2MB. This is mainly because as the cache size increases, memory traffic becomes smaller and hence the bandwidth caused by conflict L2 misses, which are not captured in our stack distance model, becomes more outstanding.

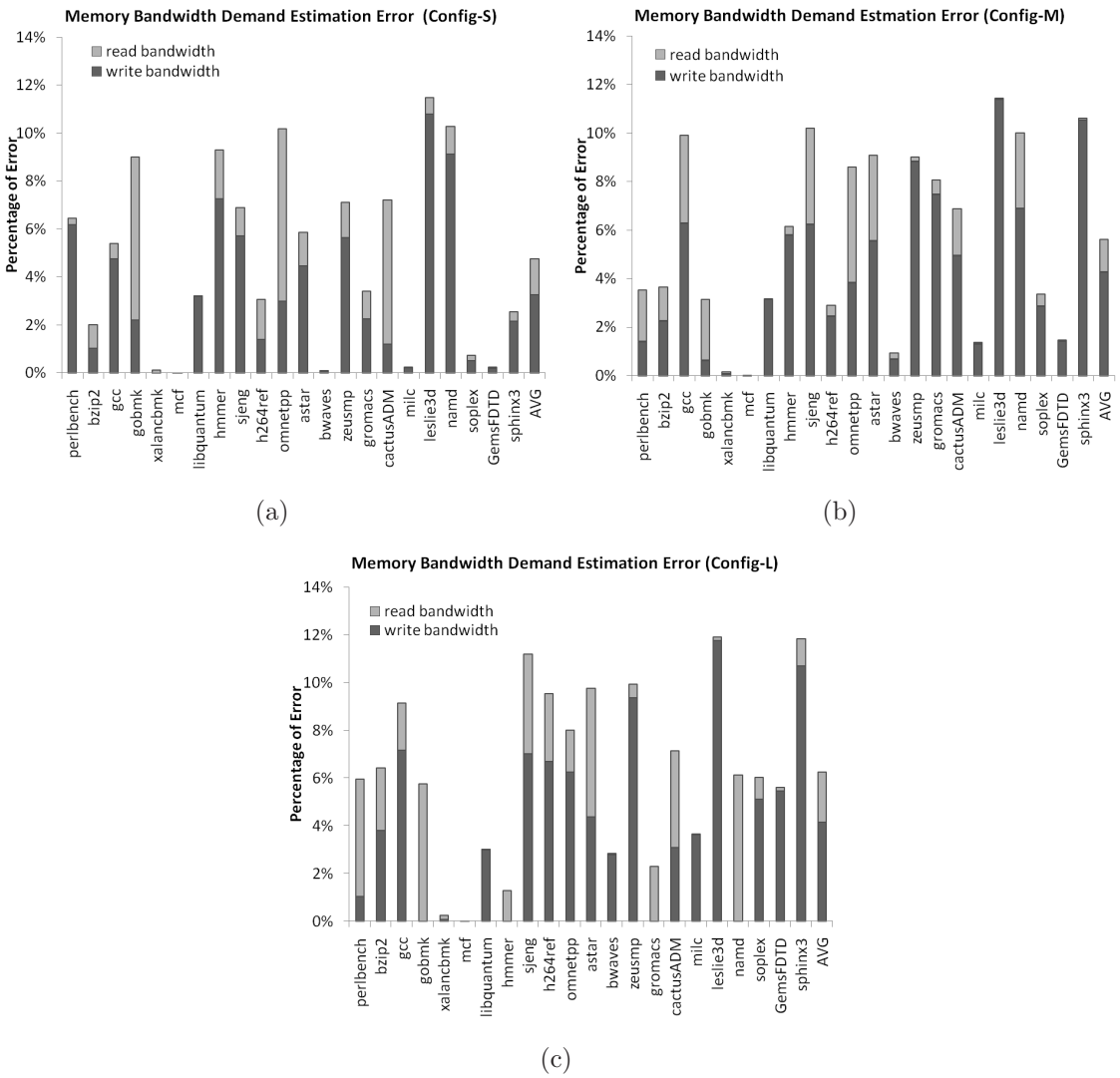


Figure 5.8: The memory bandwidth estimation error

To evaluate the demand estimation for branch predictor size, the size and the prediction accuracy of the estimated branch predictor configuration are compared with that of the largest predictor in the configuration range. The results are listed in Table 5.1. On average, by using the estimated predictor size, one could achieve 40.3% reduction in area with only 0.12% accuracy loss over the largest branch predictor.

Overall, the proposed heuristic captures the demand on branch predictor size very well.

Table 5.1: Evaluation of The Demand Estimation for Branch Predictor Size

Benchmarks	**Size Demand**		**Size Reduction**	**Accuracy Loss**
	L1 entry	**History bit**		
perlbench	1024	12	0	0
bzip2	128	12	52.5%	0.26%
gcc	1024	12	0	0
gobmk	1024	12	0	0
xalancbmk	1024	12	0	0
mcf	512	12	30%	0.06%
libquantum	8	12	59.5%	0.03%
hmmer	128	12	52.5%	0.09%
sjeng	1024	12	0	0
h264ref	1024	12	0	0
omnetpp	1024	12	0	0
astar	64	12	56.3%	0
bwaves	32	12	58.1%	2.1%
zeusmp	64	9	92.2%	0.05%
gromacs	8	7	98.5%	0
cactusADM	16	7	98.2%	0
milc	32	11	78.3%	0
leslie3d	256	12	45.0%	0
namd	128	12	52.5%	0.1%
soplex	256	12	45.0%	0.01%
GemsFDTD	16	7	98.2%	0.01%
sphinx3	1024	12	0	0
avg	-		40.3%	0.12%

5.7.2.2 Multi-Resource Demand Estimation

The quality of multi-resource demand estimation includes two aspects: the accuracy in satisfying the performance target and the energy efficiency of the estimated resources.

To evaluate the accuracy, this dissertation performed detailed simulation with the resource configurations estimated at each performance target ranging from 50% to 95%. The obtained relative performance (normalized to the largest performance

in the design space) is compared against the corresponding performance target. The differences are summarized in Figure 5.9(a). The observed error is up to 12.7% (on *soplex*), and the maximum average error is 8.6% (on *xlanacbmk*). Note that this dissertation only reports the results with performance target larger than 50% to avoid the *ill-suited* cases that some program may have a small performance variation range and its smallest relative performance may be much larger than the performance target.

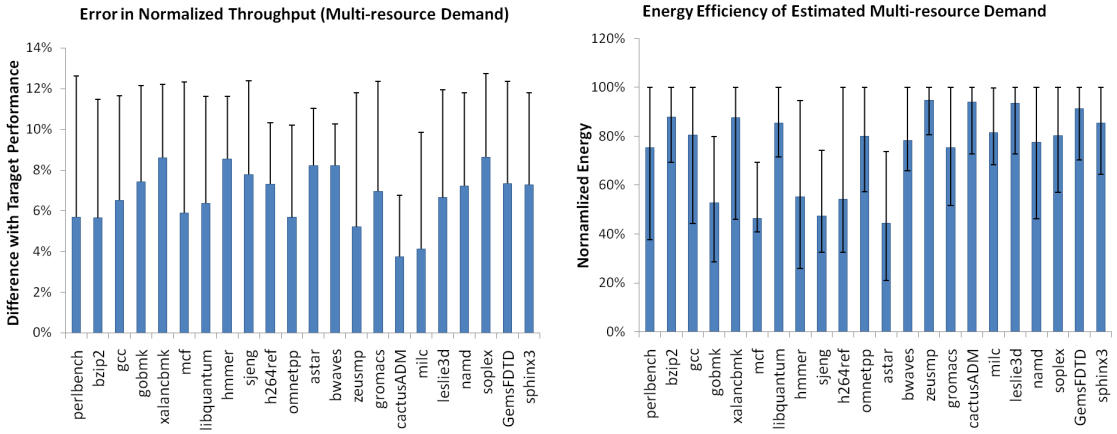


Figure 5.9: Evaluation of Multi-resource Demand Estimation. The results are based on estimating 4 different resource demands.

To evaluate the energy efficiency, this dissertation also compares the energy consumption of the estimated multi-resource demand with the energy consumption of other resource combinations that satisfy the given performance target. Due to the large design space, it is prohibitively expensive to exhaustively compare the estimated resource configurations with every eligible design node. Therefore, this dissertation uses Monte Carlo simulations to simulate 300 random samples in the design space, and group them into the buckets of (0,0.05],(0.05,0.1],..,(0.95,1] ac-

cording to their performance relative to the highest one in the design space. Within each bucket, the energy of the estimated multi-resource configuration is divided with the maximum energy of the design nodes in that bucket. These ratios indicate the energy efficiency of the estimated resource demands, and are summarized in Figure 5.9(b). On average, the ratio is no larger than 86.5%, and can be as low as 44.4%, which means the estimated multi-resources are reasonably energy efficient.

5.7.3 Complexity Analysis

The complexity of the PREDA framework involves the complexity of multi-resource demand searching algorithm and the time cost in evaluating the performance model. As explained previously, the complexity of the algorithm depends on the number of iterations required to reach the target performance. To reduce the number of iterations, the algorithm hoists the starting point of the searching process as the target performance increases. This feature allows the algorithm to avoid unnecessary search iterations and significantly speeds up the searching process. In this experiment, the algorithm converges in no larger than 12 iterations. This dissertation also compares the CPU time required to finish *one searching iteration* by using our performance model with the time required by using the state-of-the-art analytical model developed by Eyerman, *et al.* [13]. Compared with Eyerman's model, the proposed model achieves up to 40X speedup. This is mainly because every time cache size changes, Eyerman's model requires detailed cache simulation to collect cache miss and MLP information for different window sizes, whereas our model only needs to walk through the stack distance trace. Depending on the data footprint of the programs, the profiling time cost of PREDA may be larger than Eyerman's model because of the stack distance profiling. However, this is a one-time profiling cost,

and could be easily amortized by the speedup in the demand estimation process.

5.8 Summary

This chapter presents an integrated framework for program resource demand analysis (PREDA), which leverages the synergy between the performance trend modeling and marginal utility to identify the resource demand of a workload without any detailed simulation. The proposed framework is able to estimate both single-resource and multi-resource demand on an array of processor resources, ranging from the issue width, the operating frequency to the memory bandwidth. Experimental results show that the proposed framework on average provides no larger than 8.6% error to any given performance target for multi-resource demand estimation. By using the proposed performance model, the framework achieves up to 40X speedup in multi-resource demand estimation compared with that by using state-of-the-art analytical model. This proposed framework can be applied in workload capacity planning in large CMP systems as well as early stage designs space exploration. With help of on-line profiling, it can also be applied in coordinated multiple resource management for Quality-of-Service in CMP systems as well as proactive resource adaptation in reconfigurable computing.

Chapter 6

Program-core Mapping in Static SHMP

While the resource demand analysis framework is able to identify fine-grain resource demand of the application, it is not able to establish the mapping relationship between programs and cores in static SHMP because the estimated program resource demands may not completely match the core configurations. On the other hand, blindly mapping the programs to the cores leads to poor execution efficiency in static SHMP. Therefore, there is a need for an intelligent program-core mapping technique to assign the programs to the appropriate cores such that the overall execution efficiency in static SHMP is improved. To address this problem, this chapter presents a multi-dimensional matching framework that is able to identify the appropriate program-core mapping pairs by analyzing the resource demands and the core configurations in a unified space.

6.1 Framework

The idea of multi-dimensional program-core matching stems from the observation that both programs and cores can be described with a set of orthogonal characteristics. For example, the program's ILP can be described with its instruction dependency distance [51]; whereas the processor core's capability to exploit ILP can be captured with its issue width. By projecting these characteristics from both program side and core side to a unified multi-dimensional space, this

dissertation is able to visualize the correlation between the program and the core, and simplify the program-core matching. Figure 6.1 shows the proposed frame-

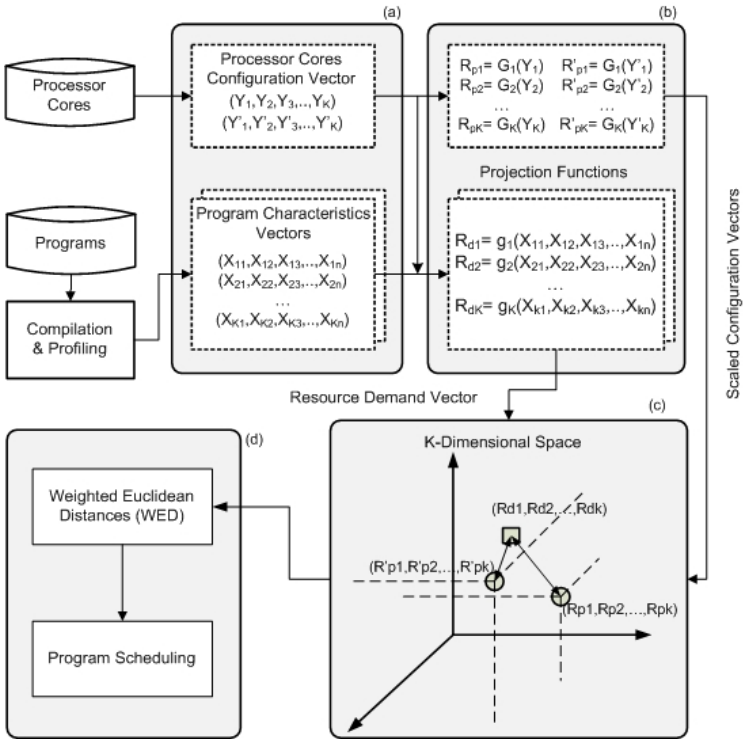


Figure 6.1: Framework for multidimensional program-core matching.

work for multidimensional program-core matching. The programs are first compiled and profiled to obtain K sets of inherent program characteristic: $\mathbf{X_1}$,$\mathbf{X_2}$, .., $\mathbf{X_K}$, where $\mathbf{X_i} = (x_{i0}, x_{i1}, x_{i2}, .., x_{in})$ is the vector that describes program characteristic $i(i = 1..K)$ (Vectors are used here since program characteristics may require more than one number to represent). These vectors are then transformed by a set of projection functions $g_i(i = 1..K)$, which transform the characteristic $\mathbf{X_i}$ to the program's resource demand $R_{di}(i = 1..K)$, as shown in Figure 6.1(b). Specifically, $R_{di} = g_i(\mathbf{X_i}) = g_i(x_{i0}, x_{i1}, x_{i2}, .., x_{in})$. Therefore, the program's Resource Demand Vector (RDV) is: $\mathbf{R_D} = (R_{d1}, R_{d2}, .., R_{dK}) = (g_1(\mathbf{X_1}), g_2(\mathbf{X_2}), .., g_K(\mathbf{X_K}))$. This vec-

tor points to the program's desired configuration node in the K-dimensional space, as shown in Figure 6.1(c). On the other hand, the configurations of each processor core can be described with a vector$(y_1, y_2, .., y_K)$, where each element y_i is in correspondence with the program characteristic $\mathbf{X_i}$ $(i = 1..K)$. Similarly, this vector can also be transformed by a set of projection functions to a scaled configuration vector $\mathbf{R_P}$ $=(R_{p1}, R_{p2}, .., R_{pk})$ in the K-dimensional space. Once both the resource demand vector $\mathbf{R_D}$ and the configuration vector $\mathbf{R_p}$ have been projected to the same space, the distance between these two becomes the natural measurement for the degree of match between the program and the core. Specifically, larger distance leads to less compatibility between the program and the core, and hence less execution efficiency. Note that not every dimension of the vector contributes equally to the degree of match, some of them are more important whereas others are less outstanding. Therefore, this dissertation uses the Weighted Euclidean Distance (WED) in the K-dimensional space as the metric for the match between the program and the core, as shown in Equation 6.1:

$$WED^2 = \sum_{i=1}^{k} w_i (R_{di} - R_{pi})^2 \tag{6.1}$$

where w_i is the weight coefficient for the $i-th$ dimension $(i = 1..K)$. Given that, the program scheduler can identify the matching program-core pairs by simply comparing the WEDs from the program to different cores.

6.2 Projection Function

Given this framework, the projection functions become the key component, as they map the core configurations and the program characteristics to the unified K-dimensional space for the given objectives. Two sets of projection functions are

employed for this purpose. One set of them is used to scale the raw hardware configurations in accordance with the diminishing return effect. The other set of them is to interpret and quantify the implications of program's inherent characteristics on its hardware resource demand.

Generally speaking, hardware resources suffer from the diminishing return effect, that is, the increase in hardware resource yields less than proportional increase in the marginal benefit. For example, issue width usually has 4 possible values: 1, 2, 4 and 8; and the diminishing return effect states that the benefit gain by increasing the issue width from 1 to 2 is the largest, followed by that from 2 to 4, and that from 4 to 8. This diminishing return effect has its implication on the program-core distance: the difference in the distance decreases as the value of core configuration increases. To capture this effect, the reciprocal function is used to scale the inter-configuration distance such that the space between adjacent configurations decreases as the configuration value increases. Therefore, the projection function for the configuration y of core i could be formulated as follows:

$$R_{y,i} = c(\frac{1}{y_{min}} - \frac{1}{y_i}) \tag{6.2}$$

where y_{min} is the minimum value of the configuration y among the cores, and y_i is the value of configuration y of core i ($i = 1..n$). The parameter c in the equation is a normalizing factor, and is used to set the value of $R_{y,i}$ in the range of [0,1]. Note that there are many other functions one could use to capture the diminishing return effect. This dissertation employs this reciprocal function because it is simple and it yields reasonably good results. While the proposed framework supports the projection of K ($K \geq 4$) different hardware configurations, this dissertation only examines four types of configurations, i.e., issue width, L1 data cache size, L1 instruction

cache, and branch predictor size. Each configuration occupies one dimension of the space and has four possible values positioned on the axis according to the projection function. Note that each dimension could have more than four possible values. This dissertation only uses four values to demonstrate the concept.

In accordance with the hardware configurations, this dissertation investigates four important program characteristics: instruction level parallelism, data locality, instruction locality and branch predictability. The projection functions for these characteristics are used to identify the program's demands on issue width, L1 data cache size, L1 instruction cache size, and branch predictor size, which are summarized in Table 6.1.

Table 6.1: Projection Functions

Resource Demands	**Projection Functions**
Issue Width	$R_{issue} = \sum_{i=1}^{4} P_{IW,i} * R_{W,i}$
Data Cache Size	$R_{Dcache} = \sum_{i=1}^{4} P_{Dstk,i} R_{DC,i}$
Instruction Cache Size	$R_{Icache} = \sum_{i=1}^{4} P_{Istk,i} R_{IC,i}$
Branch Predictor Size	$R_{branch} = \frac{\sum_{i=1}^{3} R_{B,i} * (P_{br,i+1} + P_{br,10-i}) + R_{B,4} * w * (P_{br,5} + P_{br,6})}{\sum_{i=2}^{4} P_{br,i} + \sum_{i=7}^{9} P_{br,i} + w * \sum_{i=5}^{6} P_{br,i}}$

The issue width demand is obtained by clustering the instructions into four groups according to their register dependency distances [51]. Specifically, group 1 with distance of 1, group 2 with distance of 2-3, group 3 with distance of 4-7, and group 4 with distance of 8 and larger. Each group has its most suitable issue width to exploit its parallelism, that is, issue width of 1 for group 1, issue width of 2 for group 2, issue width of 4 for group 3, and issue width of 8 for group 4. Then the mass center (or the weighted average) of the distribution on average indicates the issue width demand of the program. In Table 6.1, $P_{IW,i}$ $(i = 1..4)$ is the percentage

of instructions falling in each group, and $R_{W,i}$ $(i = 1..4)$ is the projected coordinates in the issue width dimension of the space representing the issue width from 1 to 8.

The program's demand on branch predictor size is calculated based on the branch transition rate [50], which captures the branch predictability of the program. Generally speaking, the branch instructions with extremely low or extremely high transition rate can be predicted with short history registers; and as transition rate approaches 50%, branches become harder to predict and requires longer history register to hold their patterns. Based on this observation, the transition rates are evenly divide into 10 buckets: [0, 0.1], [0.1, 0.2], .., [0.9, 1.0]. Then, the branches in the buckets [0.4, 0.5] and [0.5, 0.6] are associated with the largest branch predictor, those in the buckets [0.3, 0.4] and [0.6, 0.7] are associated with a smaller branch predictor, and so on (assuming performance monotonicity [47]). The mass center of the transition rate distribution indicates on average the program's demand on branch predictor size. In Table 6.1, $R_{B,i}(i = 1..4)$ is the coordinates in the branch predictor dimension of the space, with $R_{B,1}$ the smallest and $R_{B,4}$ the largest. The parameter w is employed to keep track of the fact that as the issue width gets wider the branch misprediction penalty also increases, and hence a larger branch predictor with higher prediction accuracy is required to compensate the increased misprediction penalty.

The demands on L1 data cache is derived based on Mattson's stack distance distribution [31]. It exploits the inclusion property of Least Recently Used (LRU) replacement policy, which states that, during any sequence of memory accesses, the contents of a cache with size k should be a subset of the contents of a cache with size $k + 1$ or larger. Let $C_{D(i)}(i = 1..4)$ be the possible L1 data cache sizes with $C_{D(1)} < C_{D(2)} < C_{D(3)} < C_{D(4)}$, and let $P_{Dstk,i}(i = 1..4)$ be the percentage

of the accesses whose stack distance is within the range of $[0, C_{D(1)}]$, $(C_{D(1)}, C_{D(2)}]$, $(C_{D(2)}, C_{D(3)}]$, $(C_{D(3)}, C_{D(4)}]$ respectively. According to the inclusion property, $P_{Dstk,1}$ can be most efficiently hold by the cache with size $C_{D(1)}$, and $P_{Dstk,2}$ can be most efficiently hold by the cache with size $C_{D(2)}$, and so on. Therefore, the mass center of the stack distance distribution indicates the program's average demand on L1 data cache size. The same is true for the demand on L1 instruction cache. In Table 6.1, $R_{DC,i}$ and $R_{IC,i}$ represent the scaled coordinate for the data cache size $C_{D(i)}(i = 1..4)$ and the instruction cache size $C_{I(i)}(i = 1..4)$ respectively. $P_{Istk,i}(i = 1..4)$ is the percentage of the accesses whose stack distance is within the range of $[0, C_{I(1)}], (C_{I(1)}, C_{I(2)}], (C_{I(2)}, C_{I(3)}], (C_{I(3)}, C_{I(4)}]$ respectively.

6.3 Weight Assignment

Now that the program's resource demands on issue width, L1 data cache size, L1 instruction cache size, and branch predictor size are available, the program can be projected to the 4D space with the $RDV(R_{issue}, R_{dcache}, R_{icache}, R_{brach})$. Therefore, WED between the program and core i in the 4D space becomes: $WED_i^2 = w_1 * (R_{issue} - R_{W,i})^2 + w_2 * (R_{dcache} - R_{DC,i})^2 + w_3 * (R_{icache} - R_{IC,i})^2 + w_4 * (R_{branch} - R_{B,i})^2$, $(i = 1..4)$. The weights in this equation reflect the amount of contribution each dimension has on the degree of match between the program and the core. To determine these weights, the EDPs of a representative program on 4 different configurations are obtained through detailed simulations. These data are then used as the training examples to find the appropriate weights such that the following error function is minimized:

$$J(\mathbf{W}) = \frac{1}{4} * \sum_{k=1}^{4} \left(\overline{E^{(k)}} - \sum_{j=1}^{4} w_j * (R_{dj}^{(k)} - R_{pj}^{(k)})^2 \right)^2 \tag{6.3}$$

$\overline{E^{(k)}}$ in this function stands for the normalized $(EDP)^2$ of the k-th training example, $R_{dj}^{(k)}$ stands for the projected resource demand of the k-th training example on j-th dimension, and $R_{pj}^{(k)}$ stands for the projected hardware configuration of the k-th training example on j-th dimension. By minimizing this error function, the weights are assigned in such a way that WED is pulled toward the normalized EDP as close as possible, which ensures WED to be the proxy of the normalized EDP. To minimize this error function, for any w_i, $(i = 1..4)$,

$$\frac{\partial J(\mathbf{W})}{\partial w_i} = 0$$

or

$$\sum_{k=1}^{4} \left(\overline{E^{(k)}} - \sum_{j=1}^{4} w_j * (R_{pj}^{(k)} - R_{cj}^{(k)})^2 \right) (R_{pi}^{(k)} - R_{ci}^{(k)})^2 = 0$$

Let $\theta_i^{(k)} = (R_{pi}^{(k)} - R_{ci}^{(k)})^2$, then,

$$w_1 * \sum_{k=1}^{4} \theta_1^{(k)} \theta_i^{(k)} + w_2 * \sum_{k=1}^{4} \theta_2^{(k)} \theta_i^{(k)} + w_3 * \sum_{k=1}^{4} \theta_3^{(k)} \theta_i^{(k)} + w_4 * \sum_{k=1}^{4} \theta_4^{(k)} \theta_i^{(k)} = \sum_{k=1}^{4} \overline{E^{(k)}} \theta_i^{(k)}$$

Expand this equation by using the four training examples:

$$\mathbf{A} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{4} \overline{E^{(k)}} \theta_1^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_2^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_3^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_4^{(k)} \end{pmatrix}$$

where,

$$\mathbf{A} = \begin{pmatrix} \sum_{k=1}^{4} \theta_1^{(k)} \theta_1^{(k)} & \sum_{i=1}^{4} \theta_2^{(k)} \theta_1^{(k)} & \sum_{i=1}^{4} \theta_3^{(k)} \theta_1^{(k)} & \sum_{i=1}^{4} \theta_4^{(k)} \theta_1^{(k)} \\ \sum_{k=1}^{4} \theta_1^{(k)} \theta_2^{(k)} & \sum_{i=1}^{4} \theta_2^{(k)} \theta_2^{(k)} & \sum_{i=1}^{4} \theta_3^{(k)} \theta_2^{(k)} & \sum_{i=1}^{4} \theta_4^{(k)} \theta_2^{(k)} \\ \sum_{k=1}^{4} \theta_1^{(k)} \theta_3^{(k)} & \sum_{i=1}^{4} \theta_2^{(k)} \theta_3^{(k)} & \sum_{i=1}^{4} \theta_3^{(k)} \theta_3^{(k)} & \sum_{i=1}^{4} \theta_4^{(k)} \theta_3^{(k)} \\ \sum_{k=1}^{4} \theta_1^{(k)} \theta_4^{(k)} & \sum_{i=1}^{4} \theta_2^{(k)} \theta_4^{(k)} & \sum_{i=1}^{4} \theta_3^{(k)} \theta_4^{(k)} & \sum_{i=1}^{4} \theta_4^{(k)} \theta_4^{(k)} \end{pmatrix}$$

As a result, the weights can be derived by the following formula:

$$\begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \mathbf{A}^{-1} \begin{pmatrix} \sum_{k=1}^{4} \overline{E^{(k)}} \theta_1^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_2^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_3^{(k)} \\ \sum_{k=1}^{4} \overline{E^{(k)}} \theta_4^{(k)} \end{pmatrix} \tag{6.4}$$

Note that the weights exhibit a universal pattern, that is, the ratios between $w1$,$w2$,$w3$ and $w4$ are approximately the same for the weights from different programs. This is because the relative importance of the four hardware aspects is typically the same across different programs as far as the energy efficiency is concerned. Since the purpose of WED is only to preserve the trend of EDP as the configuration changes, the wegiths could be assigned with the scaled value, rather than the exact value obtained from the gradient descent algorithm. Specifically, this dissertation finds that the weights for the dimension of issue width, L1 data cache size, L1 instruction cache size, and branch predictor size, can be assigned with 0.9, 0.15, 0.10 and 0.05 respectively.

6.4 Mapping Heuristic

WED indicates the degree of match between the program's resource demand and the core's hardware resource. The smaller the distance is, the better the match is, and hence the higher the execution efficiency would be. Therefore, the distance can be treated as a proxy of the program's execution efficiency on a certain core. Given that, the optimum scheduler shall minimize the total distance of the scheduled program-core pairs. However, such scheme is NP-complete ($O(n!)$ with a naive implementation), and becomes impractical for large number of programs. This section presents a scalable scheduling heuristic based on the program-core distance. As

Pseudocode 4 Distance Based Program Scheduling

```
#define P[M] /*programs in the program queue*/
#define C[N] /*processor cores*/
#define WED[M][N] /*weighted Euclidean distance array*/
if (M ≤ N) /* the number of programs is no larger than the number of cores*/
    for j ( 1 .. M)
        for i ( 1.. N)
            if (C[i] is available && min_dist > WED[j][i])
                min_dist = WED[j][i];
                core_id = i;
            end if
        end for
        MAP[j] = core_id;
        mark C[core_id] as unavailable, and reset min_dist;
    end for
else /* the number of programs is larger than the number of cores*/
    for i (1 .. N)
        for j (1 .. M)
            if ( P[j] has not been scheduled && min_dist > WED[j][i])
                min_dist = WED[j][i];
                prog_id = j;
            end if
        end for
        MAP[prog_id] = i;
        mark P[prog_id] as scheduled, and reset min_dist;
    end for
    if any core becomes available, schedule the program with
    min WED to that core from the remaining programs.
end if
```

shown in Pseudocode 4, when the number of programs in the queue is less than or equal to the number of available cores, the heuristic attempts to assign best available core to the given program. Specifically, the scheduler selects a program from the program queue on a first-come, first-served (FCFS) basis and allocates the available core with the minimum WED to that program. However, when the number of programs in the queue is larger than the available cores, the heuristic takes the opposite way: allocating the program with the minimum WED to the given core. The computational complexity of this algorithm is $O(n)$, where n is the number of the

programs to be scheduled. Note that this scheduling heuristic assumes that programs would be executing on the cores un-interruptively from start to finish; whereas in practice, programs are scheduled and executed based on time slices. However, from the perspective of finding the matching program-core pairs, these two are essentially the same. Therefore, the same idea could also be applied in the time-sliced based scheduling.

To make comparisons, this dissertation also examines the following scheduling algorithms:

Hardware Oblivious Mapping (H_OB): This scheduling scheme is unaware of the hardware substrate, and schedules the program on a FCFS basis. Specifically, the first in program queue is schedule to core 1, the second to core 2, and so on. This scheduling method is referred to as the baseline method in this study.

Min EDP Scheduling (MIN_EDP): This scheduling method attempts to schedule the programs such that the overall EDP is minimized. It assumes that the EDP of each program-core pair is known a priori, and hence sets the ***best*** case scenario of the overall EDP.

Max EDP Mapping (MAX_EDP): This scheduling method attempts to schedule the programs such that the overall EDP is maximized. It also assumes that the EDP of each program-core pair is known a priori, and provides the ***worst*** case scenario in the overall EDP.

6.5 Evaluation

This section presents the evaluation of the proposed program-core mapping framework. The simulation platform, workloads, and metrics used in this evaluation are described in Chapter 4.

To demonstrate that the WED is an appropriate proxy for the program's execution efficiency on a certain core, it is necessary to show that the WED has strong correlation with the EDP. To do so, this dissertation measures the EDP of each program on each possible processor core in the configuration space, and calculate the WED of every program-core pair. Table 6.2 shows the Pearson's correlation coefficient between WED and EDP for each benchmark program. The closer the coefficient is to 1, the stronger the correlation is. The average correlation coefficient of these programs is 0.8719, indicating strong correlation between EDP and WED. Note that for each program, this dissertation uses the input data specified along the program for EDP measurement, and use the training input dataset for WED calculation. Therefore, the coefficients in Table 6.2 not only demonstrate that WED is strongly correlated with EDP, but also show that the input dataset has insignificant influence on the WED model.

Table 6.2: Correlation Coefficient between EDP and WED

Benchmarks		Coeff.	Benchmarks		Coeff.
SPEC CPU2000INT	gcc-166	0.8408	SPEC CPU2000FP	apsi-ref	0.9470
	bzip2-source	0.8394		equake-ref	0.8674
	vortex-1	0.8944		applu-ref	0.9540
	mcf-ref	0.8758		lucas-ref	0.8371
	vpr-route	0.8292		mesa-ref	0.8291
	crafty-ref	0.7559		ammp-ref	0.9496
	twolf-ref	0.8513	MediaBench	mpeg2dec-tek6	0.9538
	eon-rushmeier	0.9053		cjpeg-monalisa	0.9423
	gzip-source	0.7946		ghostscript-titanic2	0.8659
	swim-ref	0.9299		epic-titanic3	0.9393
	mgrid-ref	0.9511		encode-clinton	0.8929

Note that SHMPs are designed to improve the efficiency of diverse workloads, as opposed to the homogeneous ones. Therefore, in order to evaluate the proposed scheduling heuristics, 200 diverse workloads are composed from the benchmark, each with four heterogeneous programs. Then the workloads are scheduled with the scheduling heuristics and the scheduling results for each program mixes are collected. Figure 6.2 shows the boxplots of the scheduling results for the workloads with four programs. According to these boxplots, the proposed distance based scheduling achieves on average 35.7% reduction in EDP, 26.5% reduction in makespan with slight reduction in energy (1.2%) when compared with the traditional hardware oblivious scheduling. Compared with MIN_EDP scheduling, the distance-based scheduling achieves less average reduction in EDP (35.7% vs 41.4%) and in makespan (26.5% vs 37.3%), but approximately similar reduction in energy (1.2% vs 2.4%) (The difference in energy reduction rate is of no statistical significance since the notches of the two boxes overlap, as shown in Figure 6.2(d)). Note that the reduction in energy and makespan is the by-product of the heuristics targeting at EDP minimization. Since the metric EDP is skewed towards clock cycles (or delays), minimizing EDP leads to higher reduction in makespan (or throughput) than in energy.

In addition, this section also evaluates the performance of the proposed mapping heuristic as the number of programs in the workload varies from one to seven. Figure 6.3 shows the normalized average EDP for all the scheduling schemes. The upper and lower bounds of the achievable EDP are set by the MAX_EDP scheduling and MIN_EDP scheduling respectively. In between are the EDPs achieved by the hardware oblivious scheduling and the distance based scheduling. The performance of the distance based scheduling is slightly off the optimum due to the imperfection of the WED model and the suboptimal nature of the heuristic. Nevertheless, it still

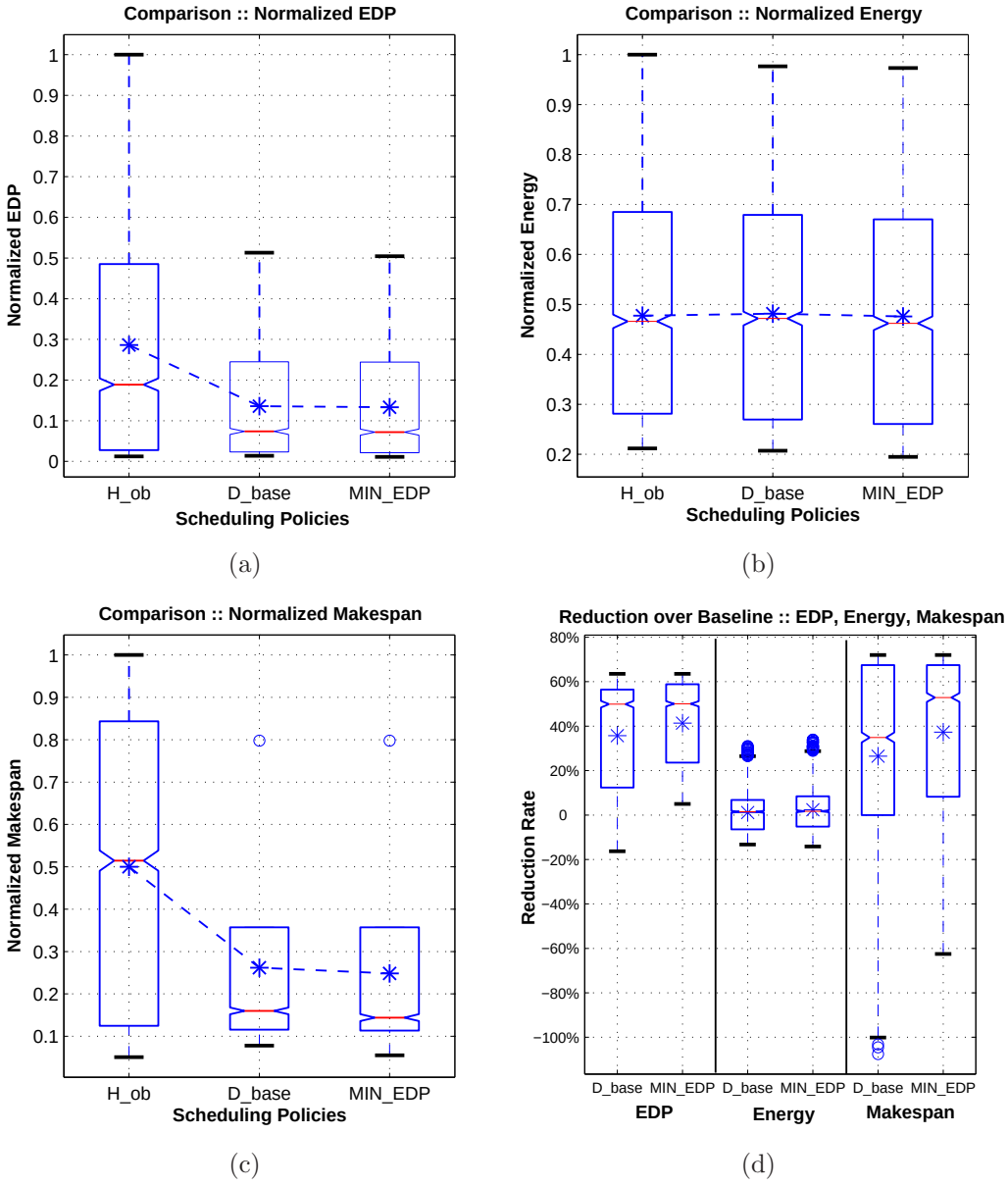


Figure 6.2: EDP, energy and makespan comparison between different scheduling heuristics. Makespan is the time between the start and finish of a group of programs, and is used as the metric for throughput [44]. The asterisk stands for the sample average.

achieves near optimum EDP on average, and its performance is consistent for all the program numbers examined in this study.

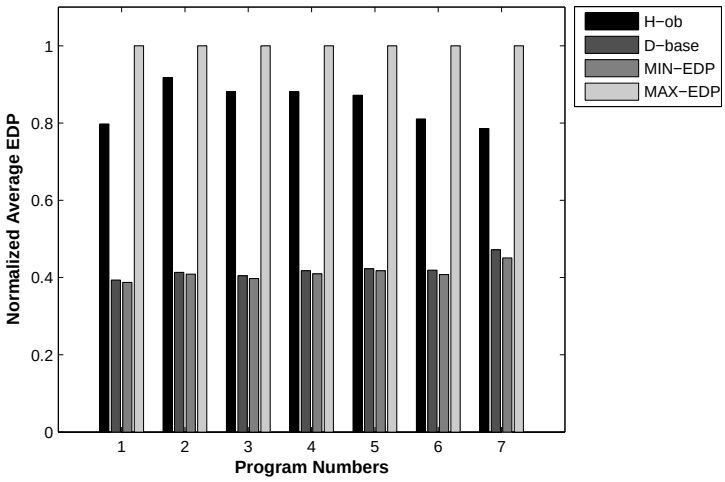


Figure 6.3: Scheduling results for different number of programs

6.6 Summary

This chapter presents a scalable framework to leverage the inherent characteristics of a program for scheduling decisions in heterogeneous multicore processors. The proposed method first transforms the program's inherent characteristics to the program's resource demand. It then projects the program's resource demands and the core's configurations to a unified multi-dimensional space, and uses weighted Euclidean distance between these two to guide the program scheduling. The experimental results show that, with four programs, our proposed scheduling heuristic achieves an average of 35.7% reduction in EDP, 26.5% reduction in makespan and 1.2% reduction in energy when compared with traditional hardware oblivious scheduling algorithm. This technique, combined with the resource demand analysis framework, provides a complete solution to the off-line heterogeneity-aware program scheduling in static SHMPs.

Chapter 7

Predictive Scheduling in Static SHMP

The program-core mapping technique presented in the previous chapter is static, and cannot adapt to program phase changes. This chapter presents a dynamic scheduling technique that is aware of the hardware heterogeneity and can adapt to the program phase changes. Unlike the existing trail-and-error scheduling, the proposed scheduling technique leverages the analytical performance model to dynamic predict the anticipated performance of the application, fundamentally eliminating the need of trial runs.

7.1 Scheduling Framework

The proposed scheduling framework, named as Predictive Heterogeneity-Aware Scheduler (PHASE), consists of three components: the on-line profiler, the performance predictor and the scheduling heuristic, as shown in Figure 7.1. The on-line profiler non-invasively profiles the application running on each core, and extracts the application's inherent characteristics required for the performance prediction. The performance predictor collects the profiled application characteristics at the end of each scheduling interval, and predicts the application's performance on other cores using the collected application statistics and the configurations of the corresponding cores. The predicted performance values are organized as a performance matrix and passed to the OS scheduler, where the scheduling algorithm identifies and enforces

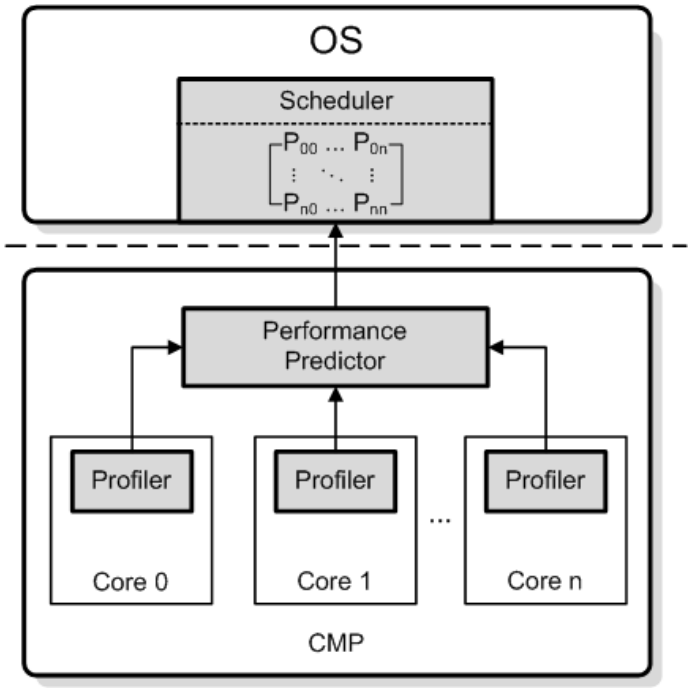


Figure 7.1: The overview of the PHASE framework.

the appropriate assignment of the applications for the next interval. As a result, the PHASE framework completely eliminates the need of trial runs, while also being able to dynamically and efficiently adapt to program phase changes.

Note that the proposed scheduling algorithm is not intended to replace, but rather complement the existing criteria for application scheduling. Specifically, the heterogeneity-aware scheduling is enforced only after the scheduler has chosen the applications from its application pool based on existing criteria including priority, fairness, and starvation-avoidance. Note also that although this framework can be applied in single-ISA heterogeneous CMP caused by design, this dissertation focuses its application on heterogeneous processors resulting from process variations and manufacturing defects. The following sections explain each component of the scheduling framework in detail.

7.2 Performance Modeling

The performance model used in this scheduling framework is described in Chapter 3. Note that this work focuses on the scheduling issue for single-threaded execution, therefore, the performance model does not need to model the impact of co-executing threads. However, in this work, the functional units in each core may not be sufficient due to manufacturing defects, the performance model does need to capture the impact of limited functional units. Refer to Chapter 3 for detailed discussions of this model.

7.3 Online Profilers

The proposed performance model requires a set of program characteristics from which the key parameters used in the performance model can be derived. These characteristics include: a) the critical dependency chain, for deriving the average ILP; b) the instruction ready set size histogram, for calculating the effective issue width with different FU number; c) the stack distance histogram [31], for estimating the number of L2 load misses with different L2 cache sizes. This section presents a set of non-invasive and cost-effective online profilers to dynamically extract these characteristics during the application's execution.

7.3.1 Critical Dependency Chain Profiler

The critical dependency chain refers to the longest instruction dependency chain in the instruction window. To capture the length of the critical dependency chain, this dissertation proposes a token-passing technique inspired by Fields, *et al.*'s work [48]. A token is a field in each reservation station entry that keeps track

of the dependency chain length, as shown in Figure 7.2(a). When an instruction enters the reservation station, its token field is set to zero; when an instruction leaves the reservation station for execution, its token field is incremented by one. The incremented token is propagated along with the result tag of the instruction. When the instruction finishes execution and its result tag matches the source tag of the waiting instruction in the reservation station, the propagated token compares the token of the waiting instruction. The larger one between these two is stored in the token field of the waiting instruction. As a result, by the time an instruction is ready for execution, its token holds the length of the longest dependency chain for this instruction.

The critical dependency chain profiler compares the token of every issued instruction, and keeps track of the maximum observed token, which is then used as an index to the max dependency chain histogram. The histogram is controlled by an instruction counter that monitors the number of issued instructions. When this number reaches the size of instruction window, the histogram entry indexed with the maximum observed token is incremented by 1. Meanwhile the register that holds the maximum token is reset to zero. Consequently, the maximum dependency chain histogram holds the information of the longest dependency chain length for each instruction window. At the end of each scheduling interval, this histogram is used to calculated the average length of the critical dependency chains, and then reset to zeros for the next scheduling interval.

7.3.2 Ready Set Size Profiler

The ready set size profiler takes advantage of the standard instruction selection logic [52], where the information about the number of ready instructions on

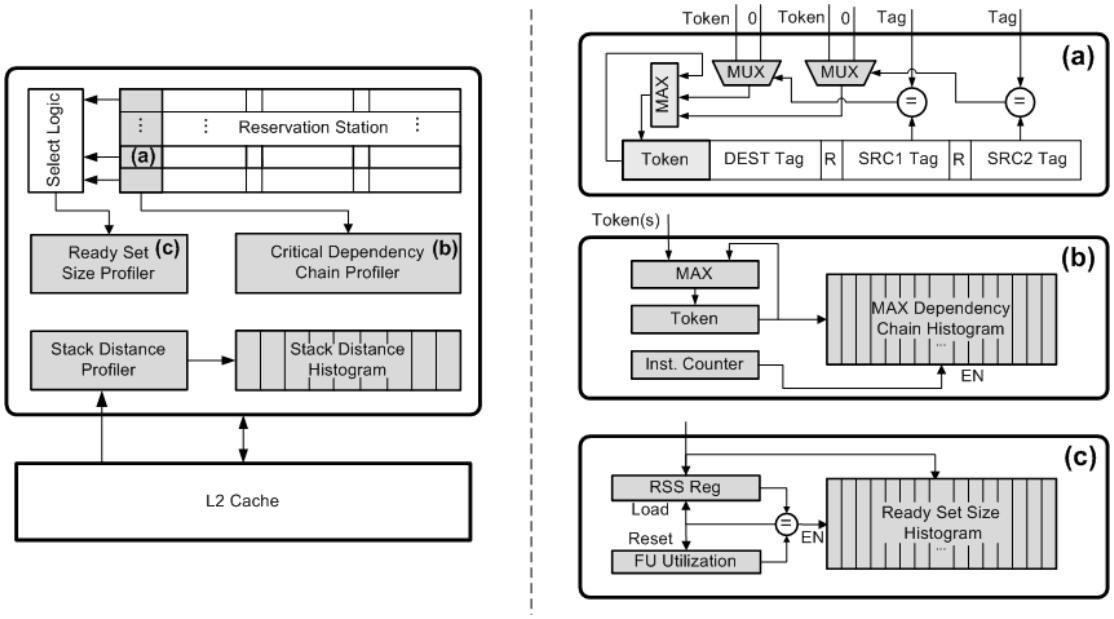


Figure 7.2: The structure of the online profilers.

a certain type of functional units is readily available. This information is steered to the RSS register in the ready set size profiler for the corresponding FU type, as shown in Figure 7.2(c). Besides the RSS register, the ready set size profiler also contains a utilization counter that is incremented each time an instruction is issued to the corresponding FU for execution. When the utilization of the FU equals the previously stored RSS value, the RSS register is loaded with a new RSS value, and the utilization counter is reset to zero. Meanwhile, the RSS histogram entry indexed by the new RSS value is incremented by one. At the end of scheduling interval, RSS histogram is used to update the average instruction latency, and reset to zero.

Such profiling mechanism can precisely capture the ready set size information assuming that instructions are issued in the *oldest-first* order. However, for a different instruction selection policy, the profiled RSS histogram may not exactly reflect the application's RSS statistics. Nevertheless, the expected discrepancy is small and its

impact on the accuracy of performance prediction is negligible.

7.3.3 Stack Distance Profiler

The stack distance profiler is used to keep track of the program memory accesses, and obtain the stack distance distribution/histogram of the program. As mentioned in Chapter 3, stack distance refers to the distance between the Most Recently Used (MRU) position and the position of a data block when it gets reused. This information allows us to estimate the number of misses at any cache sizes without any trial runs [31].

The profiling for the stack distance histogram requires an Auxiliary Tag Directory (ATD) and an array of hit counters, one for each cache way [25]. The ATD has the same associativity as the largest L2 cache in the chip and keeps track of LRU data replacement. Each time when there is a hit in the ATD, the stack distance of the hit data block is used to index into the array of hit counters, and the corresponding counter is incremented by one. Therefore, the stack distance histogram is simply composed of these hit counters organized from MRU to LRU positions. To reduce the hardware overhead caused by ATD, this dissertation employs the Dynamic Set Sampling (DSS) technique, which essentially uses a few sets to approximate the entire cache behavior [53][25]. This dissertation uses 1-to-32 set sampling ratio.

7.3.4 Profiling for Other Parameters

Other parameters required by the performance model can be obtained from the standard performance counters. For example, the performance counters in Intel® Core™ architecture [54] are able to provide the instruction mix and cache hit/miss statistics. With these statistics, the average latency lat_{avg} can be derived by weight-

averaging the percentage of each instruction type with the corresponding execution latency. Note that the load that misses L1 cache but hits in L2 cache is treated as an instruction with long execution latency. This average latency is further adjusted with the RSS histogram to count in the effect of limited functional units.

Similarly, the factor for average memory level parallelism m_{ovp} can be obtained by monitoring the *Miss Status Holding Register* (MSHR) in L2 cache. Specifically, every time an L2 load miss happens, MSHR is queried for outstanding load misses. m_{ovp} is the average number of these outstanding misses across all L2 load misses.

7.3.5 Hardware Cost Analysis

The hardware cost of the profilers depends on the instruction window size as well as the L2 cache size. Assuming 128 instruction window size, 96-entry reservation station, 32-bit physical address space, and 2MB 8-way L2 cache with 64B block size, the total hardware cost amounts to 3.5KB, as shown in Table 7.1. The hardware cost may be further reduced by using a smaller number of histogram counters based on the observation that most of the ready set size or the critical dependency chain length is far smaller than the instruction window size. However, even without such optimization, the online profilers incur no larger than 0.2% hardware overhead on a core with 2MB L2 cache. Note that these profilers are not in the critical path, and do not affect the application's maximum performance.

The computation cost of the performance prediction is mainly caused by converting the profiled histograms to the parameters used in the performance model, which would require about 300 multiply and accumulate operations. In addition, the performance model itself needs 2 add, 1 comparison, 2 multiply and 3 divide

Table 7.1: Hardware Cost of the Online Profilers

Profiler	Components	Costs
Critical Dependency Chain Profiler	token fields	7*128 bits
	multiplexors, comparators	(7*2+7)*96 bits
	histogram counters	32*128 bits
RSS Profiler	RSS Reg & Utilization Reg.	7*2 bits
	histogram counters	32*128 bits
Stack Distance Profiler	LRU bits per ATD entry	3 bits
	valid bits per ATD entry	1 bits
	address bits per ATD entry	12 bits
	total ATD cost (with 128 sampled sets)	(3+1+12)*8*128 bits
	Hit Counters	32*(8+1) bits
Cost of Profilers Per Core		27790 bits

operations. Therefore, predicting an application's performance on four cores involves approximately 350 arithmetic operations. Since the prediction is made only once every scheduling interval, these operations can be performed on the functional units already on the chip by stealing their idle slots. By starting computing the estimated performance several thousands of cycles before the end of the scheduling interval, the computation can be completely hidden and will not incur performance penalties.

7.4 Scheduling Heuristics

To identify the optimum application-core allocation from the performance matrix, a naïve approach requires exhaustive search, which has the complexity of $O(n!)$ and is not scalable. In contrast, our PHASE scheduler uses a greedy algorithm with polynomial computation complexity, as shown in Pseudocode 5. The algorithm first searches the estimated performance matrix[1] for the largest entry, and the corresponding program and core indices are stored in the application-core al-

[1]Depending on the optimization target, the performance in the matrix could be in the form of IPC or IPC speedup.

location array and then removed from the index arrays. This process repeats for the remaining matrix until all indices of applications or cores have been visited. The newly obtained application-core allocation is enforced in the next scheduling interval only when the predicted performance gain is larger than the given *migration threshold*. This threshold serves as a *migration throttling* agent, which prevents applications from migrating when there is insufficient performance improvement to compensate the migration cost. The impact of the threshold value is evaluated in Section 7.5.2.

Pseudocode 5 Algorithm for Predictive Heterogeneous-Aware Scheduling

```
#define N_c /*the number of cores in the CMP*/
#define N_p /*the number of programs to be scheduled*/
#define P_th /*the performance threshold*/
#define perf[N_p][N_c] /*the array of predicted performance*/
#define prog[N_p] /*the program index array*/
#define core[N_c] /*the core index array*/
#define core_alloc[N_c] /*the core allocation array*/

for ( i=0; i < N_c; i++ )
    foreach n_c in core[N_c]
        foreach n_p in prog[N_p]
            if ( perf[n_p][n_c] > max_perf )
                N_pmax = n_p; N_cmax = n_c;
                max_perf = perf[n_p][n_c];
            end if
        end foreach;
    end foreach;
    core_alloc[N_cmax] = N_pmax;
    total_perf = total_perf + max_perf;
    remove N_pmax from prog[N_p];
    remove N_cmax from core[N_c];
end for
if ((total_perf - total_monitored_perf)/total_monitored_perf > P_th)
    enforce schedule based on core_alloc[N_c];
end if
```

The complexity of this algorithm is $O(n^2 \cdot m)$, where n is the number of cores

and m is the number of programs to schedule ($m \leq n$). Note that if the number of programs is larger than the number of cores, the scheduler will first choose the programs from the program pool using the criteria such as priority and fairness, and then the performance matrix associated with these programs will be searched for the optimum allocation. This dissertation focuses on the case that the number of programs is no larger than the number of cores.

Besides this proposed algorithm, this dissertation also evaluates a set of other scheduling algorithms for comparison. These algorithms include:

OpenSolaris: This is the default OpenSolaris scheduler, which is unaware of the core-level heterogeneity and treats each core as symmetric. It has the property of natural binding, that is, when an application gets scheduled to one core, this application is unlikely to be migrated to a different core in the next scheduling interval to avoid migration overhead [55]. Therefore, it can be treated as random static mapping, and used as the baseline scheduler in this work.

Becchi+: This algorithm is based on the one proposed by Becchi, *et al.* [6]. While the original proposal only applies to two types of cores, this dissertation extends it to support four or more different cores. Specifically, the algorithm allows the applications run for one interval, and then it randomly selects two cores, swaps the applications running on the cores, and makes them run for another interval. The allocation that gives the higher aggregated throughput between these two intervals is enforced in the next scheduling interval. The procedure repeats in the next scheduling interval.

Oracle: This algorithm assumes the application's performance on different cores in the next scheduling interval is known *a priori*. It uses these *future* performance data

to find the application-core allocation that gives highest throughput (or speedup), and enforce the allocation in the next scheduling interval. While it is unrealistic in practice, it sets an upper bound of the potential performance improvement.

Worst Static Scheduling (WSS): This is the static application-core mapping that gives the lowest aggregated throughput (or speedup). It is only used as a reference point to highlight the worst situation that an heterogeneity-unaware scheduling scheme could possibly end up with.

7.5 Evaluation

This section presents the evaluation of the proposed application scheduling framework. It consists of the evaluation of the model accuracy, the choice of migration threshold, and the performance improvements over other scheduling schemes. The simulation platform, workloads, and metrics used in this evaluation are described in Chapter 4.

7.5.1 Model Accuracy

The accuracy of the performance model could largely impact the effectiveness of the proposed scheduling framework. To evaluate the model accuracy, every SPEC CPU2006 program is run on a simulated processor for one scheduling interval, and then the performance model is used to estimate the program's CPI on target processors with different configurations. Meanwhile, these programs are also simulated on those target processors for one scheduling interval and the observed CPI is compared with the estimated one. As shown in Figure 7.3(a)-(c), the average error between the estimated CPI and the observed one is no larger than 8.17%, indicating the per-

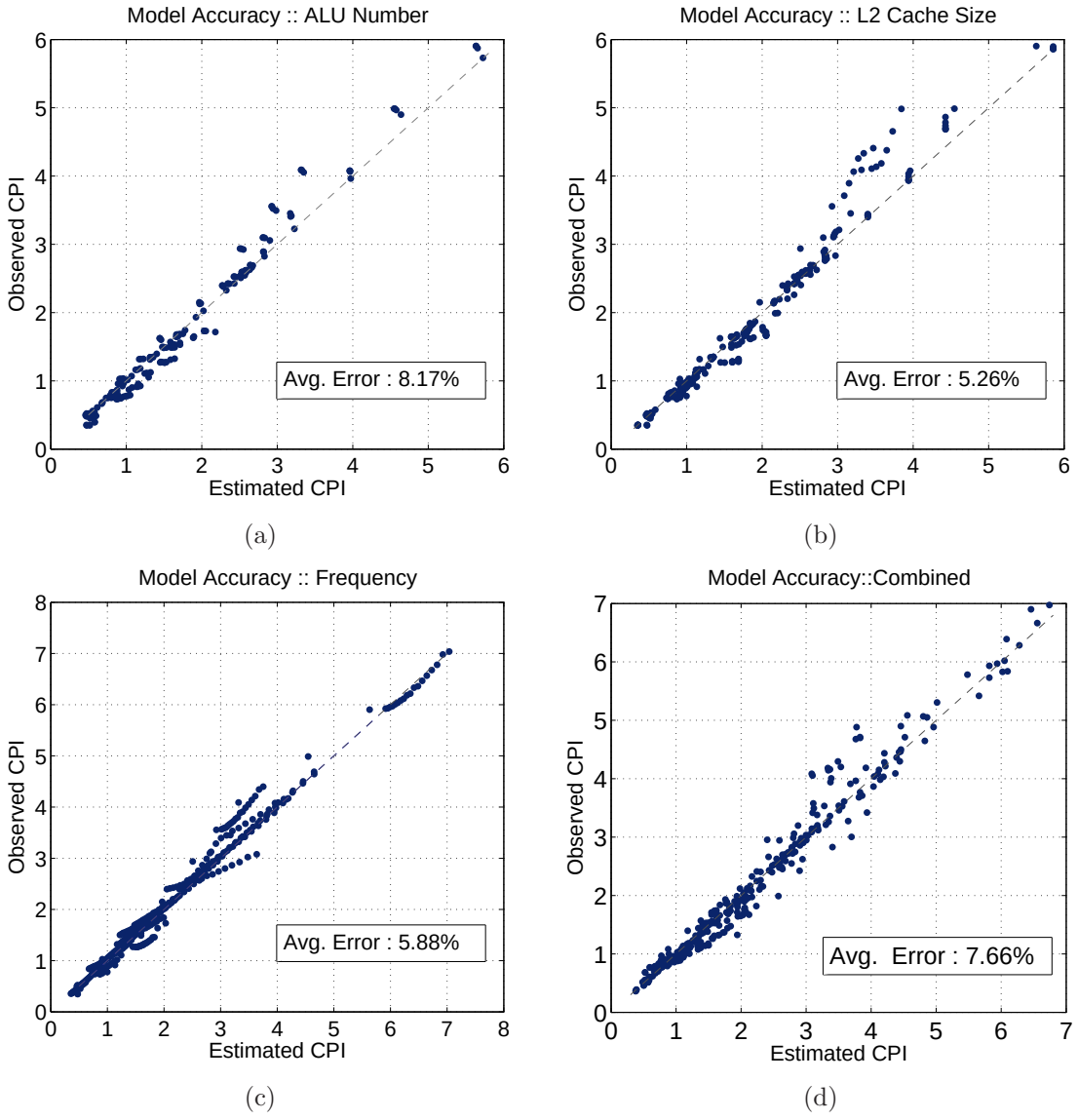


Figure 7.3: Model Accuracy. (a) The number of IALU varies from 1 to 4. (b) The L2 cache size varies from 512KB to 2MB at the step of 256KB. (c) Frequency varies from 2GHz to 4GHz at the step of 0.1GHz. (a)-(c) only one resource changes with others in nominal configurations. (d) 300 random configurations when three sources vary simultaneously.

formance model keeps track of the observed performance very well when only one resource varies its configuration. Figure 7.3(d) shows the Monte Carlo simulation of 300 random configurations when all three resources vary simultaneously. The aver-

age error between the estimated CPI and the observed one is 6.71%, and the largest error is 22.7%. It is also observed that the relative error follows normal distribution, indicating the analytical model contains little systematic error.

7.5.2 Migration Threshold

As explained in the previous section, migration threshold is used in the proposed scheduling algorithm as an important parameter to control the performance gain and throttle non-beneficial program migration. Figure 7.4 shows the performance and migration frequency of workload *lxws* at different migration thresholds. As the migration threshold increases from 0, the number of migration goes down while the overall throughput improves. This is because the increase of the migration threshold filters out the detrimental application migrations whose migration overhead is larger than the potential performance improvement. On the other hand, a large threshold could also cause performance degradation. As shown in the figure, the performance drops sharply when the threshold increases beyond 7%. This is because a high threshold conservatively prevents the application from migrating, filtering out the opportunities for performance improvement. Therefore, a good threshold should prevent most of the detrimental application migrations yet still allow most beneficial application migrations. This work uses 5% as the threshold value for all workloads. One can further improve the performance by using an adaptive threshold, but it is beyond the scope of this dissertation.

7.5.3 Performance

Throughput Improvement: Figure 7.5(a) shows the comparison of the aggregated throughput for different scheduling policies. One can observe that the

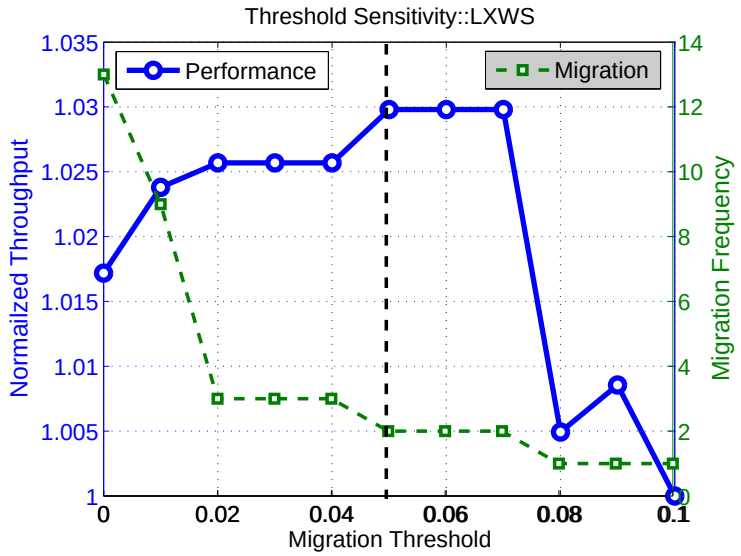


Figure 7.4: Migration Threshold. The dark dash line highlights the threshold used in this dissertation.

performance of the *OpenSolaris* scheduler can be very close to (e.g., workload *mbpg*) or significantly higher (e.g., workload *xnlo*) than that of the *WSS* scheduler. This is because the *OpenSolaris* scheduler does not consider the underlying hardware heterogeneity, and the random nature of application-core assignment may end up with a reasonable good static assignment or the worst static assignment. This also means that a scheduler that is unaware of the core-level heterogeneity may lead to non-deterministic performance, which further underscores the importance of heterogeneity awareness in application schedulers. This figure also shows that *Becchi+* scheduler has a significant improvement over the baseline *OpenSolaris* scheduler, yet its performance is still far from that of the *Oracle* scheduling. It is mainly because of the inability to quickly identify the optimum application-core assignment with explorative trial runs. In contrast, the proposed predictive scheduling scheme eliminates the trial runs and can achieve near optimum performance improvement. On average,

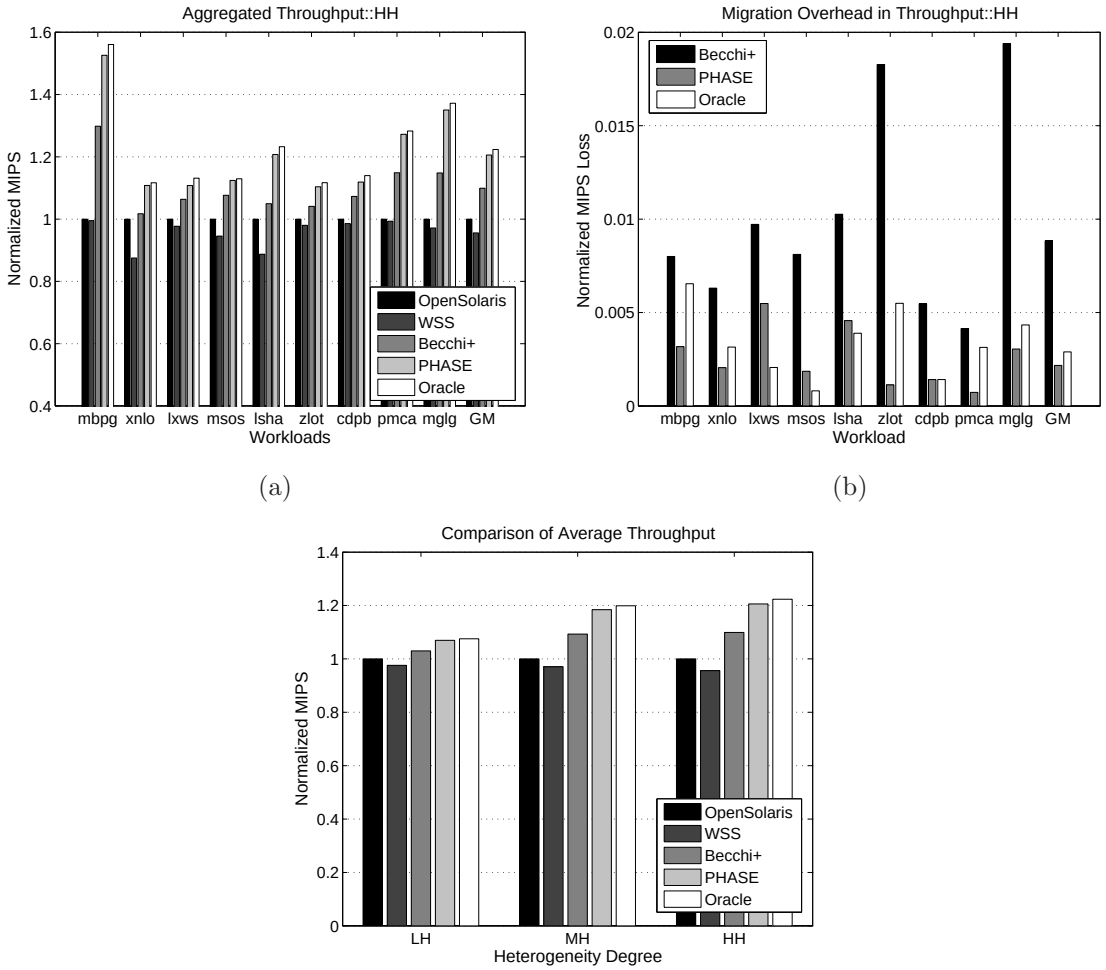


Figure 7.5: Comparison of Throughput. The data are normalized to the throughput of the OpenSolaris scheduler.

it achieves 20.6% improvement over the baseline, 11.2% improvement over *Becchi+*, and is only 1.7% less than the oracle scheduling. The small sub-optimality mainly comes from two sources: a) PHASE uses the history information to estimate future performance, hence cannot capture the sudden phase change in the next scheduling interval; whereas the *Oracle* scheduler knows the future events, and can adjust the scheduling decisions accordingly; b) due to the greedy nature of the searching algo-

rithm, PHASE may be trapped in finding the application assignment that is only locally optimum whereas the Oracle scheduler always enforces the global optimum assignment. Figure 7.5(b) illustrates the impact of migration overhead on the system throughput. It is obtained by comparing the realistic throughput with the throughput achieved when the data working sets are ideally moved along with the migrating application. It can be observed that the migration overhead of *Becchi+* is consistently the largest for each workload. This is mainly because *Becchi+* requires trial runs to determine the application assignment, causeing many unnecessary movement of data sets and slowing down the overall execution. Figure 7.5(c) shows the average throughput (geometric mean) improvement as the heterogeneity degree changes. One can also observe that the potential of the throughput improvement drops as the heterogeneity degree decreases. This intuitively makes sense because with reduced heterogeneity, the performance difference of scheduling an application to different cores is also reduced.

Efficiency Improvement: Figure 7.6(a) shows the comparison of the efficiency in terms of $mips^3/W$ for different scheduling algorithms. It is observed that PHASE achieves 3.2X efficiency improvement on workload *mbpg* compared with the baseline scheduling. This improvement is because *OpenSolaris* scheduler blindly assigns the memory-bound *mcf* to the fastest core (C-0) and the computing-bound *gcc* to the slowest core (C-3), whereas PHASE schedules the programs in the opposite way, resulting high efficiency improvement. On average, PHASE improves the efficiency by 71.6% over the *OpenSolaris* scheduler and 36.2% over *Becchi+* scheduler. Note that for some workloads, e.g., *mbpg*, *WSS* scheduling yields higher efficiency than the baseline scheduling, indicating that baseline scheduling may consume more energy than *WSS* scheduling. Figure 7.6(b) shows the efficiency loss caused by mi-

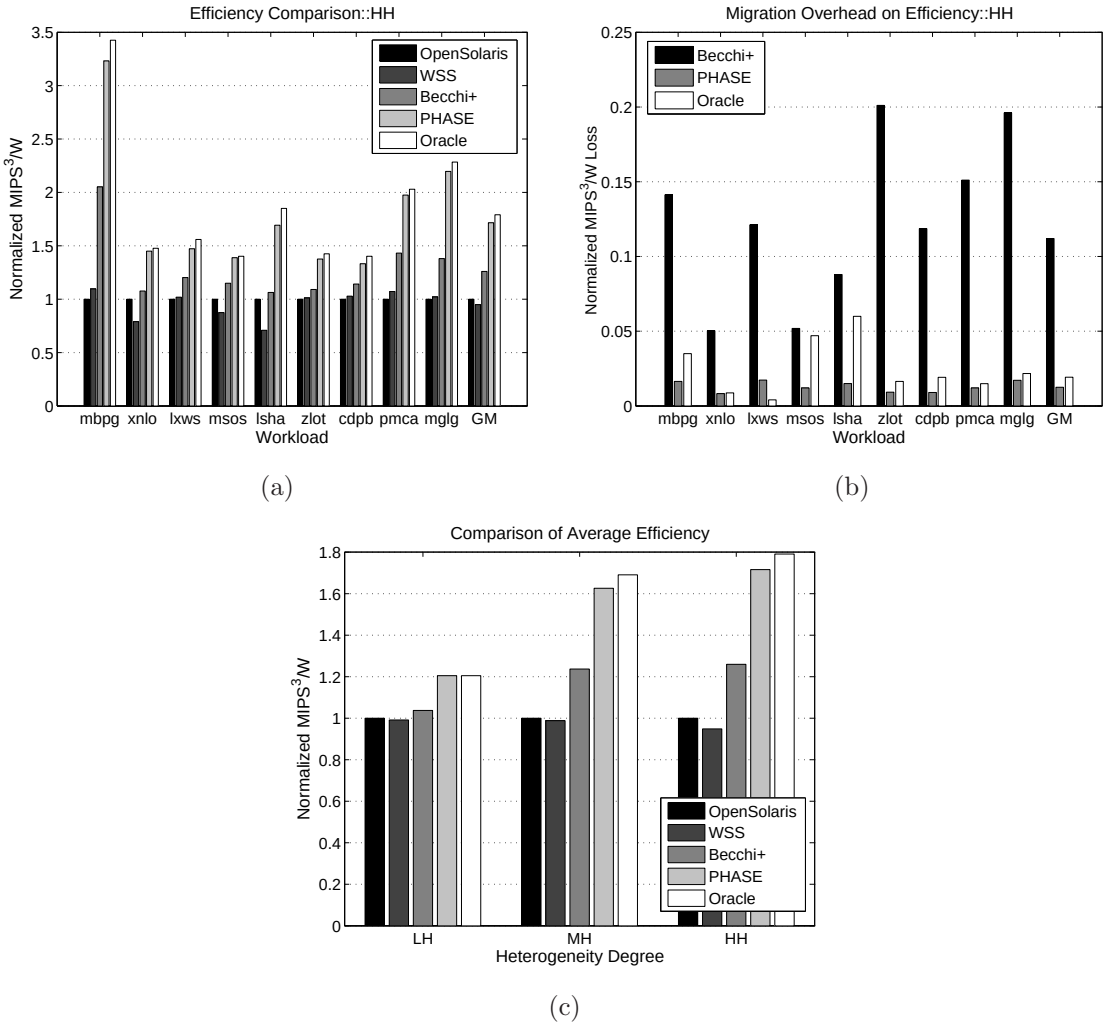
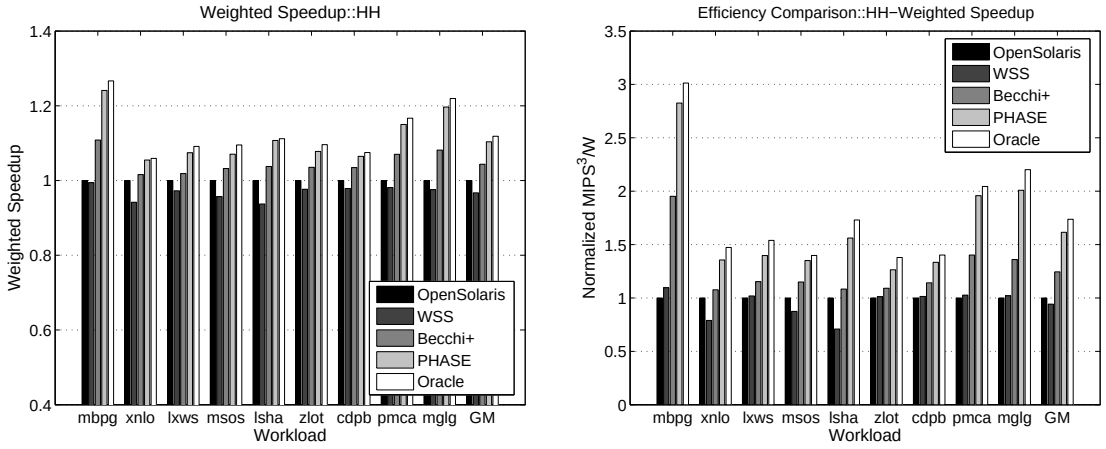


Figure 7.6: Comparison of Efficiency. The data are normalized to the efficiency of the OpenSolaris scheduler.

gration overhead. Again, *Becchi+* has the highest efficiency loss because the trial runs not only slow down the execution but also incur extra power consumption on the interconnection network between caches. Figure 7.6(c) further shows the efficiency improvement as the heterogeneity level changes. Similar to the throughput, the potential of efficiency improvement decreases as the heterogeneity degree decreases.

Weighted Speedup Improvement: Figure 7.7 shows the performance and efficiency of different schedulers when using the weighted speedup as the optimization target. The results are similar with those of the aggregated throughput, yet with smaller improvements. On average, PHASE improves the weighted speedup by 11.3% and the $mips^3/W$ efficiency by 58.6% over *OpenSolaris* scheduler, and compared with *Becchi+* scheduler, the improvements are 6.8% and 25.9% respectively.



(a) (b)

Figure 7.7: Comparison of weighted speedup and efficiency.

Impact of Program Number: This dissertation also evaluates the impact of program number on the performance of the schedulers. To do so, for each of the 4-programmed workloads, all possible combinations of 1, 2 and 3 programs are evaluated. The geometric means of the throughput results are shown in Figure 7.8(a). It can be observed that compared with the baseline scheduling, the performance of *Becchi+* decreases as the program number drops from 4 to 1. It is mainly because the scheduler unaware of the heterogeneity is more likely to reach a good static application assignment as the program number gets smaller. However, the performance of our predictive scheduling is still near optimum, and can reach up to 14.5% im-

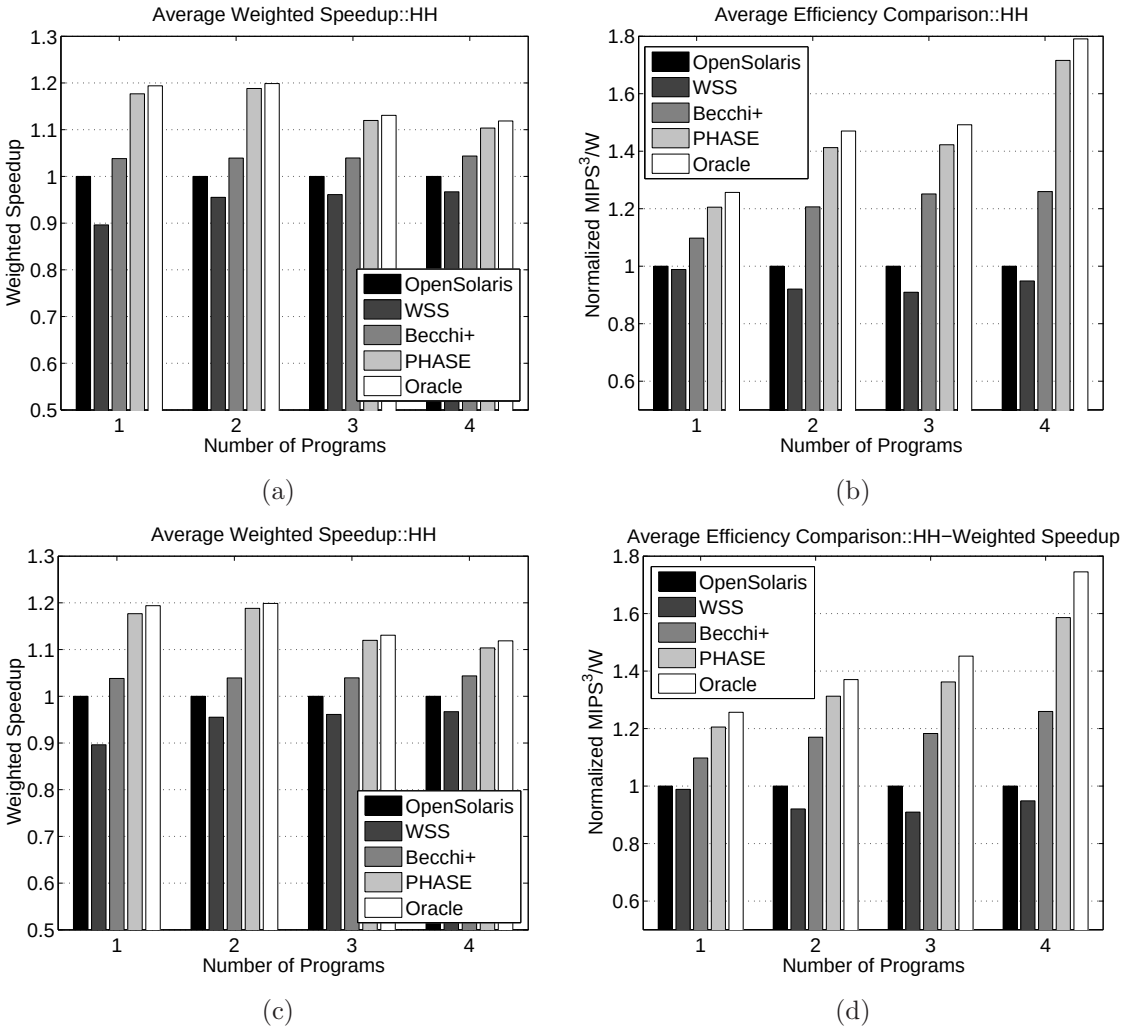


Figure 7.8: Average performance and efficiency improvement vs program number.

provement over *Becchi+*. Figure 7.8(b) further shows the efficiency improvement as the program number changes. Overall, the potential of efficiency improvement decreases as the number of program decreases. Figure 7.8(c) and (d) show the results of the same experiment as Figure 7.8(a) and (b), but with weighted speedup as the optimization target.

7.6 Summary

This chapter presents PHASE, a heterogeneity-aware scheduling framework that can dynamically and pro-actively schedule applications in single-ISA heterogeneous CMPs. This framework uses a set of hardware-efficient online profilers and a performance model to simultaneously predict the application's performance on different cores. Based on the predicted performance, the scheduler identifies and enforces near-optimal application assignment for each scheduling interval, eliminating the need of trial runs or off-line profiling. Experimental results show that the proposed heterogeneous-aware scheduler improves the commodity OpenSolaris scheduler by an average of 20.6% in terms of overall throughput and an average of 71.6% in terms of efficiency. Compared with the state-of-the-art research scheduler, the proposed scheduler improves the throughput by an average of 11.2% and the efficiency by an average of 36.2%. All of these performance gains are achieved with only a few kilobyte of additional hardware. This predictive scheduling scheme fundamentally avoids the inefficiencies and shortcomings of the existing heterogeneity-aware scheduling schemes, hence is an attractive solution to the scheduling problem in static SHMPs.

Chapter 8

Predictive Resource Coordination in Dynamic SHMP

The previous chapter addresses the issues of efficient application scheduling in static SHMP. However, when it comes to dynamic SHMP, multiple hardware resources can be dynamically allocated to the executing programs. Therefore, the problem becomes how to manage multiple interacting resources to achieve energy efficient computing and enforce Quality-of-Service (QoS) performance objectives. This problem becomes even more challenging when each core support Simultaneous Multi-threading (SMT). Under such circumstance, the resource sharing in a Chip-Multiprocessor (CMP) is compounded with both inter-core and intra-core resources, and any resource management scheme without coordinating between these two types of resources could lead to suboptimal system performance and inability to enforce system performance objectives.

As an example, Figure 8.1 shows the comparison of the weighted speedups for different combination of inter-core and intra-core resource management schemes in a quad-core 2-way SMT CMP system. Inter-core resource here is represented by L2 cache, and intra-core resources include issue queue (IQ), reorder buffer (ROB), and physical registers, all partitioned in proportion to each other [21]. As one can see, although separate management of L2 cache or intra-core resources improves the performance over the scheme of equal partition, it still misses a large amount

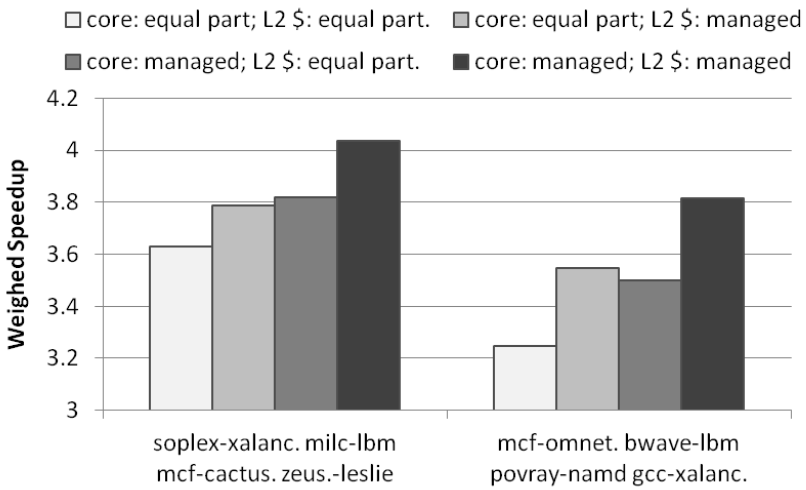


Figure 8.1: Comparison of weighted speedup for different resource management policies. Results are based on a quad-core CMP with per-core 2-way SMT (Detailed configurations in Table 4.3).

of potential for improving system performance compared with the one that coordinates the distribution of L2 cache and intra-core resources. This is because the application's demands on different resources are correlated, and the change of the application's intra-core resource allocation could affect the its demands on inter-core resources. For example, the increase of ROB size may expose more memory level parallelism (MLP) and increase the number of outstanding load misses. Since multiple load misses could hide the latency with each other, the average cache miss penalty is reduced, hence the requirement of L2 cache size is smaller in order to maintain the same performance. Therefore, coordinating between intra-core and inter-core resources is necessary to achieve high utilization and system performance in CMP+SMT environment.

However, existing management schemes for multiple interacting resources focus on either intra-core resource partitioning for a single-core SMT processor or

inter-core resource allocation for a chip-multiprocessor. Moreover, existing schemes for multiple resource management often rely on trial runs, which is inefficient in terms of performance and energy. To address these limitations, this chapter presents a comprehensive yet cost-effective resource management framework that can coordinate both intra-core and inter-core shared resources meanwhile simultaneously enforce QoS performance objectives. Unlike the existing resource management schemes, the proposed framework leverages an analytical performance model to predict the performance, and enforces resource allocations without any trial resource partitioning or training. By using the application characteristics dynamically collected during the application's execution, the performance model can update the performance prediction at each resource adaptation epoch, allowing the resource allocation to dynamically adapt to program phase changes.

8.1 Resource Coordination Framework

The proposed framework for multiple resource management mainly consists of three different components: the on-line profilers, the performance predictor, and the resource allocator, as shown in Figure 8.2. The on-line profiler non-invasively profiles each thread running on each core, and extracts the inherent characteristics of the thread for the performance prediction. The performance predictor collects the profiled characteristics of the thread at the end of each resource allocation epoch, and estimates the performance of the thread for different resource allocations. The resource allocator uses a built-in search engine to identify the appropriate resource allocations under the constraint of the given QoS targets, and enforces the resource partition for each thread through a set of partition knobs.

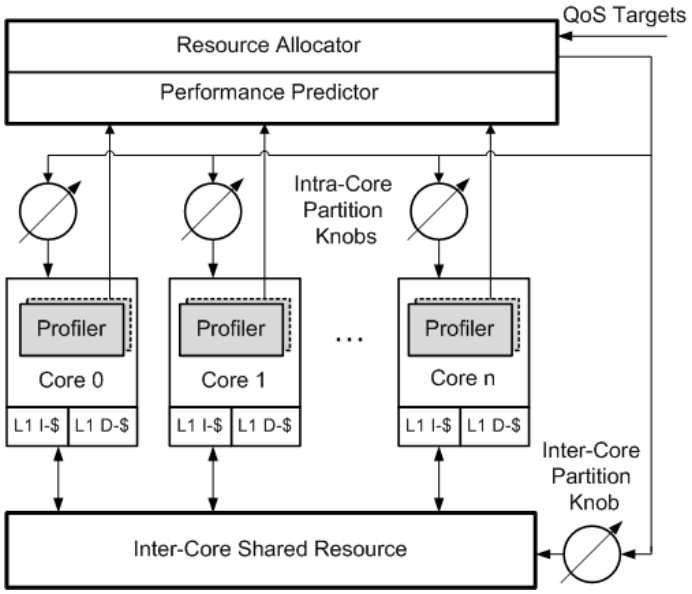


Figure 8.2: The overview of the multiple resource management framework.

The intra-core partition knobs regulate the allocation of the intra-core resources, which include Instruction Queue (IQ), ROB, and physical registers. These resources are interdependent, and are allocated *in proportion to* each other, similar with the way employed in the work by Choi, *et al.* [21]. On the other hand, the inter-core partition knobs control the distribution of Last Level Cache (LLC) size and the power consumption of each core. This dissertation assumes that the CMP uses L2 cache as LLC and supports per-core Dynamic Voltage and Frequency Scaling (DVFS). In DVFS, the voltage and frequency are correlated, hence the power management can be achieved by controlling the operating frequency of each core meanwhile keeping the total power within the budget. This framework does not explicitly manage the memory bandwidth. Instead, it uses PAR-BS memory scheduling policy [40] to ensure the fairness and QoS of bandwidth usage.

While this framework addresses the resource allocation issues in the CMP+SMT

scenario, it could also be applied in the cases where each core only supports single thread but can be dynamically reconfigured. Nevertheless, this dissertation focuses on the CMP platform with each core supporting 2-way SMT to demonstrate the effectiveness of the framework. The following sections explain each component of the proposed framework in detail.

8.2 Performance Prediction

The performance predictor predicts the performance of an application under different resource allocations by using the analytical performance model described in Chapter 3. Since this work assumes the functional units in the core are sufficient, the performance model in this work does not consider the impact of limited functional units. However, the impact of memory level parallelism (MLP) and co-executing threads have to be carefully modeled for accurate performance prediction. Specifically, for a given application, the number of non-overlapped L2 load misses is affected by two factors: the L2 cache size, which determines the total number of L2 load misses, and the ROB size, which controls the amount of exposed MLP. Therefore, when both ROB size and L2 cache size can be reconfigured, their compounded effect has to be modeled in order to estimate the number of non-overlapped L2 load misses. While Chapter 3 describes an off-line technique to estimate the non-overlapped L2 load misses under such scenario, this chapter presents a slightly different technique which is based on the same idea but more suitable for on-line implementation.

This technique uses the *load histogram* to hold the statistics of the number of loads occurred within a certain ROB size. Specifically, each time when the number of

Pseudocode 6 Non-overlapped L2 Load Miss Estimation

```
#def N_l //maximum number of loads in the ROB size i
#def N_novp //number of non-overlapped L2 load misses
#def MLP_i //average load MLP rate in ROB size i
#def ld_miss_rate //L2 load miss rate
#def ld_hist_i[N_l] //load histogram for ROB size i

for ( j=0; j < N_l; j++ )
    if (j * ld_miss_rate < 1)
        temp = ld_hist_i[j] * j * ld_miss_rate;
    else
        if (j * ld_miss_rate/MLP_i < 1)
            temp = ld_hist_i[j];
        else
            temp = ld_hist_i[j] * j * ld_miss_rate/MLP_i;
        end if
    end if
    temp_novp = temp_novp + temp;
end for
N_novp = ceiling(temp_novp);
```

retired instructions equals the given ROB size, the number of loads observed in those retired instructions is used as an index to the load histogram, and corresponding entry in the load histogram is incremented by one. With the load histogram, one is able to model the "*window*" effect the ROB has on the non-overlapped L2 load misses by estimating the outstanding misses only with the loads occurred in the ROB. The details of the technique is illustrated in Pseduocode 6. By using a set of load histograms with each dedicated to a certain ROB size, the "window" effect of different ROB sizes has been taken care of for the estimation of the non-overlapped L2 load misses. On the other hand, the L2 load miss rates for different L2 cache sizes can be estimated with the stack distance model, which is explained in Chapter

3, section 3.2.3.

Figure 8.3 shows the accuracy of the estimation technique for program *libquantum* under different ROB and L2 cache sizes. There is a close match between the measured and the estimated non-overlapped L2 load misses when both ROB size and L2 cache size vary. This dissertation also validates this technique using other SPEC CPU2006 programs, and it is observed that the average error rate of the estimation is 12.2%. Most of the errors are caused by the artifact that a small number of L2 load misses leads to a large relative error even though the absolute difference between the measured and the estimated is small. However, since a small number of L2 load misses means a small impact on the overall CPI, the influence of the estimation error passed down to the estimated CPI is also insignificant.

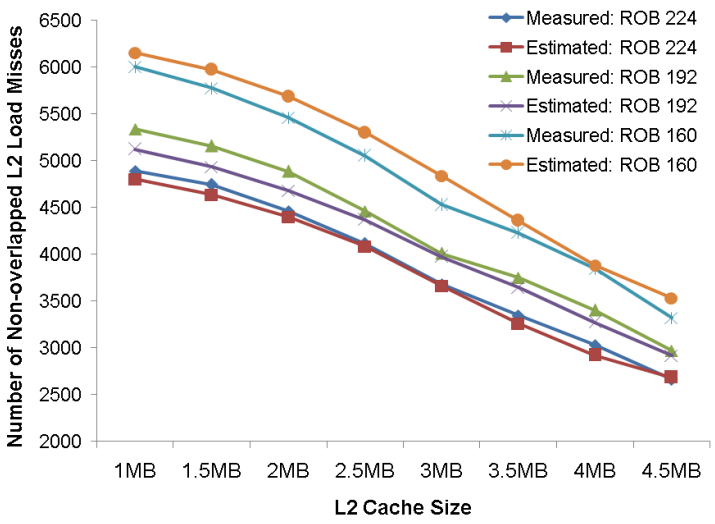


Figure 8.3: The comparison of estimated and measured non-overlapped L2 load misses for SPECCPU2006 program *libquantum*. Data are collected at an interval of 2M instructions.

8.3 On-line Profiling Support

The proposed performance model requires a set of program characteristics from which the key parameters for the model can be derived. These characteristics include: a). the critical dependency chain, for deriving the average ILP; b). the dependent load miss statistics, for estimating the MLP under different ROB sizes; c). the stack distance statistics [31], for estimating the number of L2 load misses with different L2 cache sizes. This section presents a set of non-invasive and cost-effective online profilers to dynamically extract these characteristics during the application's execution. Note that the stack distance profiler used to get the stack distance statistics is same as the one described in previous chapter, hence will not be discussed again in this chapter.

8.3.1 Critical Dependency Chain Profiler

As shown in Figure 8.4, the critical dependency chain profiler is similar with the one proposed in the previous chapter. However, in order to obtain the dependency chain length for different ROB sizes, a set of critical dependency chain histograms are required, with one histogram dedicated to one specific ROB size. All histograms share one instruction counter to count the number of issued instructions. When the number equals one of the interested ROB sizes, the corresponding histogram is updated, and the counter continues counting until it equals the largest ROB size. Then, the counter is reset and starts counting from zero again. In this way, the token fields designed to profile for the largest ROB size can be reused by multiple histograms for different ROB sizes.

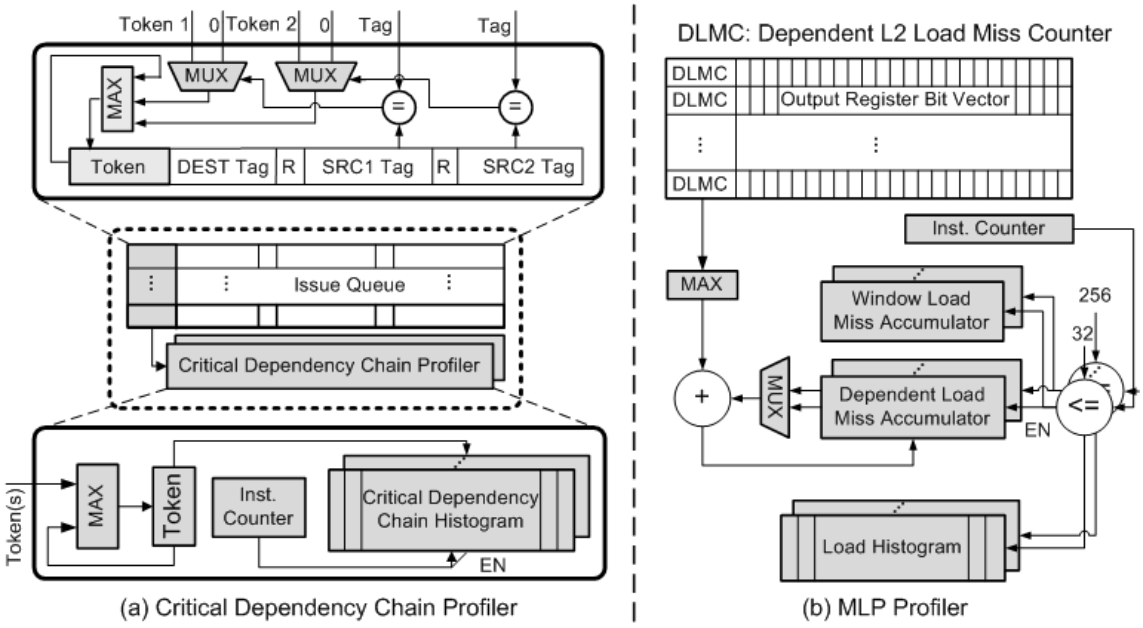


Figure 8.4: The structure of the online profilers.

8.3.2 MLP Profiler

The MLP profiler is to capture the L2 load miss parallelism for different ROB sizes. As shown Figure 8.4(b), this profiler contains a L2 *Load Miss Event Table* (LMET), which has a *Dependent Load Miss Counter* (DLMC) and a *Output Register Bit Vector* (ORBV) in each table entry, similar with the one proposed by Eyerman and Eeckhout [32]. Each time a load that missed L2 cache is retired, a new entry in the table is created and the corresponding DLMC is updated with the number of L2 load misses that this load is dependent on in the current window. Meanwhile, the ORBV is initialized by setting '1' to the bit indexed by the output register ID of this load, and setting '0' to the remaining bits. Each retired instruction thereafter needs to check its dependency on this long-latency load by looking up the ORBV bit at the position corresponding to the input register ID of the retired instruction. A '1' in this bit position indicates this instruction depends on the previous long-latency

load, and hence the bit indexed by the output register ID of the retired instruction is also set to '1'; whereas a '0' means this instruction is independent with the previous long-latency loads, and no further actions is needed. This process continues until the number of analyzed instructions reaches the largest ROB size of interest, in this dissertation, 256, and then the table is reset.

Besides the table, the profiler also has a set of Dependent Load Miss Accumulators (DLMA) and Window Load Miss Accumulators (WLMA), one for each possible ROB allocation. Each time when the analyzed instruction number equals an interested ROB size R, the associated DLMA is accumulated with largest DLMC in the table. Meanwhile, WLMA is also accumulated with the number of L2 load misses occurred among the analyzed instructions. Therefore, at the end of each epoch, the average load MLP rate of ROB size R can be obtained by dividing the values between the corresponding WLMA and DLMA pair.

In addition, the profiler has a load histogram for each possible ROB size. The histogram collects the number of loads occurred in each ROB window, and is used to estimate the non-overlapped L2 load misses.

8.4 Resource Coordination Algorithm

With the online profilers and the performance predictor, the performance of the application under different resource allocations can be estimated by simply evaluating an equation, which fundamentally eliminates the need of trial runs and can significantly improve the quality and efficiency of multiple resource management.

To efficiently manage multiple resources, this chapter presents a predictive and coordinated resource management algorithm that leverages the performance

Pseudocode 7 Coordinated Predictive Hill-Climbing(CPHC)

```
#def N_tt //total number of threads
#def N_res //the number of resources independently partitioned
#def delta //resource partition granularity
#def P_th //convergence threshold
#def part[0 : N_tt][0 : N_res] //the resource partition array
#def max_id(A, n) //get the index of the largest value in A[0:n]
#def max(A, n) //get the largest value in A[0:n]
#def perf_eval(part)
//estimate the overall performance for resource array part
#def perf(part, i)
//estimate the performance of thread i for resource array part

old_part_perf = perf_eval(part);
copy part[0 : N_tt][0 : [N_res] to temp_part[0 : N_tt][0 : N_res];
while(TRUE)
    for( i = 0; i < N_res; i++)
        for( j = 0; j < N_tt; j++ )
            temp_part[i][j] = part[i][j] + delta;
            pos_perf[j] = perf(temp_part, j);
            temp_part[i][j] = part[i][j] − delta;
            neg_perf[j] = perf(temp_part, j);
        end for
        pos_tid[i] = max_id(pos_perf, N_tt);
        neg_tid[i] = max_id(neg_perf, N_tt);
        if(max(pos_perf, N_tt) > max(neg_perf, N_tt))
            part[pos_tid[i]][i] = part[pos_tid[i]][i] + delta;
            part[neg_tid[i]][i] = part[neg_tid[i]][i] − delta;
        end if
    end for
    new_part_perf = perf_eval(part);
    if ( abs(new_part_perf − old_part_perf) < P_th) break;
    else old_part_perf = new_part_perf;
end while
```

predictor to identify the optimum resource distribution for the workload. As shown in Pseudocode 7, the proposed algorithm uses *hill-climbing* to search for the appropriate

resource distribution. Specifically, it first uses the performance model to evaluate the performance of each thread as one of the resources is incremented or decremented by a certain amount *delta*. It then moves *delta* amount of the resource from the thread that has the lowest performance degradation to the thread that benefits most from the additional resource, provided that the overall performance gain is positive. This process iterates through different resources, and repeats itself until the estimated performance reaches the given target or no noticeable performance gain is attainable. In this way, this algorithm explores the resource allocation in the positive-gradient direction, and hence achieves fast convergence.

In this algorithm, power as a resource is *indirectly* managed by controlling the operating frequency of each core in a CMP. Specifically, for a quad-core CMP, the total power consumption can be written as $a_1v_1^2f_1+a_2v_2^2f_2+a_3v_3^2f_3+a_4v_4^2f_4$, where v_i and $f_i(i = 1..4)$ are the voltage and frequency of core i respectively, and $a_i(i = 1..4)$ is the product of the activity factor and the effective capacitance for core i. In a fully-loaded CMP system, the power is usually consumed as close to the given power budget as possible to maximize performance, and $a_1, .., a_4$ are generally very close to each other. Therefore, the problem of power management can be transformed to the problem of allocating frequencies such that $v_1^2f_1 + v_2^2f_2 + v_3^2f_3 + v_4^2f_4$ remains constant. Note that the frequency and voltage are dependent under DVFS, and for a given frequency, the corresponding voltage can be found by looking up a table. Therefore, by controlling the frequencies, the power can be allocated the same way as other resources.

Besides this proposed algorithm, this dissertation also evaluate a set of other resource allocation algorithms for comparison, which include:

Equal Partition: This algorithm distributes all shared resources equally among the threads. Specifically, the inter-core resources are equally partitioned for all active threads in the CMP, and the intra-core resources are equally partitioned for the simultaneously executed threads in the core. This algorithm is used as the baseline management scheme in this dissertation.

Coordinated Reactive Hill-Climbing (CRHC): Like the proposed predictive scheme, this algorithm also attempts to manage both intra-core and inter-core resources, but without a performance prediction model. Therefore, it has to rely on trial runs to explore the gradient direction for resource allocation. Specifically,the algorithm randomly selects two threads (for inter-core resource) or a pair of co-executing threads (for intra-core resource), tentatively moves *delta* amount of resource from one thread to the other, and runs the workload for one epoch. It then moves the resource in opposite direction for the two threads, and runs the workload for another epoch. The resource allocation that gives the higher performance during these two trial runs is enforced in the next epoch. The process keeps on repeating itself for different resources and different threads.

Intra-core Reactive Hill-Climbing (Intra-RHC): This algorithm is similar with the one proposed by Choi, *et al.* [21]. The resource adaptation only happens on the intra-core level, and the inter-core resources are equally partition for all threads.

Inter-core Reactive Hill-Climbing (Inter-RHC): This algorithm is similar with CRHC except that the resource adaptation only happens on the inter-core level, and the intra-core resources are equally partition for the co-executing threads in the core.

Oracle: This algorithm assumes the application's performance under different resource allocation in the next epoch is known *a priori*. It uses these *future* performance

data to enforce the resource allocation that gives highest performance in the next epoch. While it is unrealistic in practice, it sets an upper bound of the potential performance improvement.

8.5 Implementation Cost Analysis

Both the on-line profilers and the resource allocator are implemented in hardware, and they are the major sources of the implementation cost in the proposed framework. The cost of the profilers depends on the ROB size, the L2 cache size, the number of SMT threads, as well as the partition granularity. Assuming a 256-entry ROB with 32-entry partition granularity, 160 issue queue size, 32-bit physical address space, 16MB 32-way shared L2 cache, and 2-way SMT, the total hardware cost amounts to approximately 22KB, as shown in Table 8.1. The hardware cost may be further reduced by using a smaller number of histogram counters based on the observation that the critical dependency chain length is far smaller than the ROB size. However, even without such optimization, the hardware overhead incurred by the online profilers only amounts to 0.14% of the 16MB L2 cache size. Note that these profilers are not in the critical path, and does not affect the application's execution.

On the other hand, the cost of the resource allocator is mainly caused by converting the profiled histograms to the parameters for the performance model and searching for the appropriate resource allocation with the performance model. For example, to obtain the average critical dependency chain length from the dependency chain histogram, approximately 300 multiply-add operations are required. To further quantify the hardware cost, the resource allocator is also implemented in Verilog HDL, and synthesized into a netlist. The design employs pipelining so that arithmetic

Table 8.1: Hardware Cost of the Online Profilers

Profiler	**Components**	**Costs**
Critical Dependency Chain Profiler	token fields	8*256 bits
	multiplexors, comparator	(8*2+8)*160bits
	histogram counters	16*256*8*2bits
MLP Profiler	LMET	(4+32)*16*2bits
	DLMA	16*8*2 bits
	WLMA	16*8*2 bits
	comparators	8*8*2 bits
	load histogram	16*256*8*2 bits
Stack Distance Profiler	valid bits per ATD entry	1 bits
	addr. bits per ATD entry	12 bits
	total ATD cost (32 sampled sets, 2 threads)	(3+1+12)* 32*32*2 bits
	Hit Counters	16*32*2 bits
Total Cost of Profilers per core		21568 Bytes

units can be reused. Overall, it has two adders, two multipliers and one divider, all in 32-bit fixed-point. The total area of the resource allocator is estimated to be 0.632 mm^2 under 65nm technology. Each performance estimation requires 20 cycles to complete, and the search process takes less than 30000 cycles before it converges (convergence is enforced if the iterations is larger than 20). Since the resource allocation is made only once every epoch, the latency can be completely hidden by starting resource exploration procedure several thousands of instructions before the end of the epoch.

8.6 Evaluation

This section presents the evaluation of the proposed resource management framework. It consists of the evaluation of the model accuracy, the performance of the proposed resource management scheme, and the effectiveness of QoS enforcement.

The simulation platform, workloads, and metrics used in this evaluation are described in Chapter 4.

8.6.1 Model Accuracy

The accuracy of the performance model could largely impact the effectiveness of the proposed resource management framework. To evaluate the model accuracy, every SPEC CPU2006 program is run on a simulated processor for an interval of 2 million instructions, and use the performance model to estimate the program's CPI on target processors with different resource configurations. Meanwhile, the program is also simulated on those target processors for the same interval and the observed CPI is compared with the estimated one. As shown in Figure 8.5(a)-(c), the relative error between the estimated CPI and the observed one follows normal distribution. The average errors (using absolute values) are 8.7% for different ROB sizes, 5.3% for different L2 cache sizes, and 6.7% for different frequencies, indicating the performance model keeps a good track of the observed performance when only one resource varies its configuration. Figure 8.5(d) further shows the relative estimation error for 500 random configurations when all three resources vary simultaneously. The average CPI estimation error in this scenario is 8.1%, and the largest one is 26.7%. It is also observed that the relative error follows normal distribution.

8.6.2 Epoch Size Sensitivity

The epoch size determines the frequency of resource adaptation during the execution of the workload, and can indirectly influence the overall performance of our resource management framework. Figure 8.6 shows the performance trend of three workloads as the epoch size increases from 500 kilo to 5 million instructions.

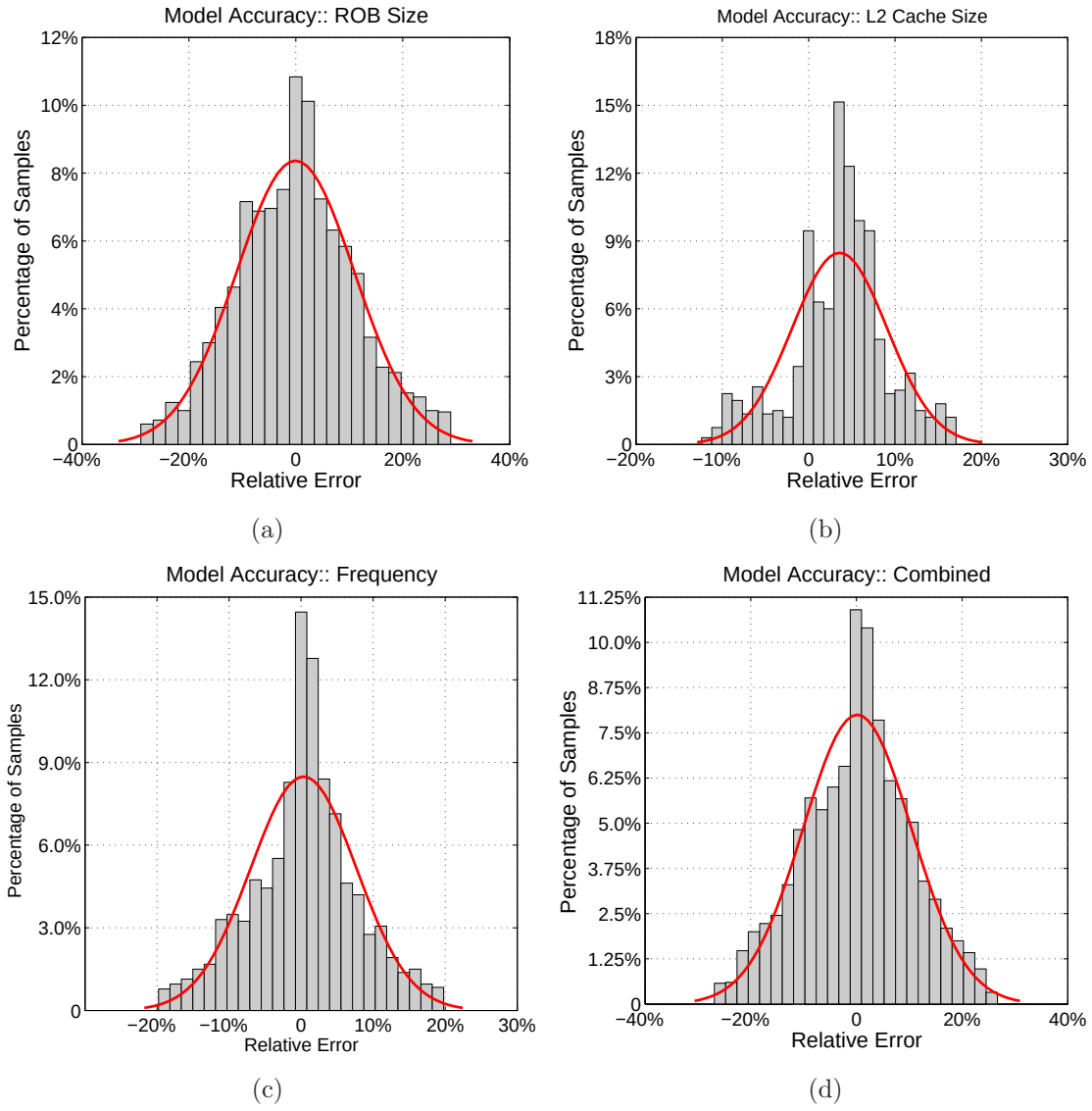


Figure 8.5: Performance Model Accuracy. (a)The ROB size varies from 32 to 256. (b)The L2 cache size varies from 512KB to 4MB at the step of 512KB. (c) Frequency varies from 2GHz to 4GHz at the step of 0.1GHz. (d) 500 random configurations when all three resources vary simultaneously.

It is observed that as the epoch size increases, the weighted speedup first increases, then reaches a plateau, and then gradually decreases. This is because with a relatively small epoch size, the on-line profilers may not be fully warmed up to capture

the corresponding program characteristics, which could affect the accuracy of the performance predictor, and in turn pulls down the performance of the resource management. This is particularly true for the stack distance profiler since this profiler employs set sampling technique, which provides a good accuracy only when it has be exercised with sufficient amount of L2 accesses. On the other hand, a large epoch size would miss the opportunity for adapting resource distribution to some finer grain program phases, which also degrades the end performance. In this work, 2 million instruction is considered to be a reasonable epoch size that balances the accuracy of the performance predictor and the responsiveness of the resource allocation.

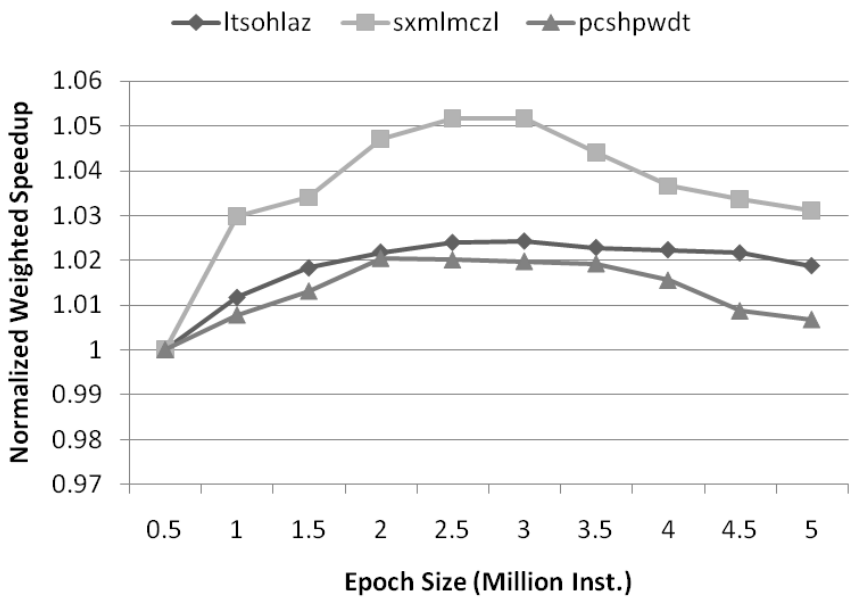


Figure 8.6: Performance impact of epoch size. The workloads *ltsohlaz,sxmlmczl,pcshpwdt* are described in Section 4.2.4.

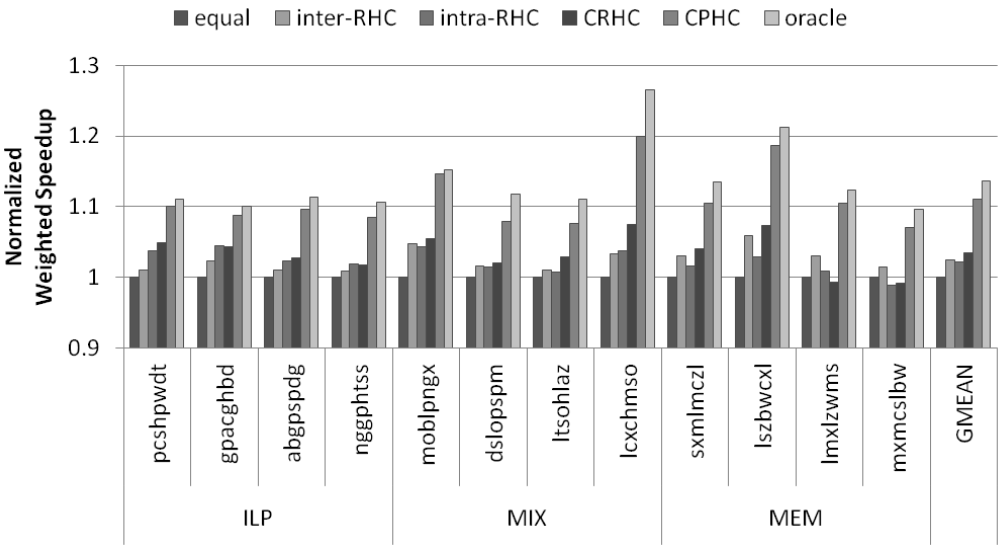
Note that such choice of epoch size is based on the assumption that different voltage and frequency pairs can be enforced instantaneously. In practice, this is not true because it may take the voltage regulator hundreds of micro-seconds to stabilize

voltage. Under such circumstance, the epoch size need to incorporate this additional time for voltage regulation.

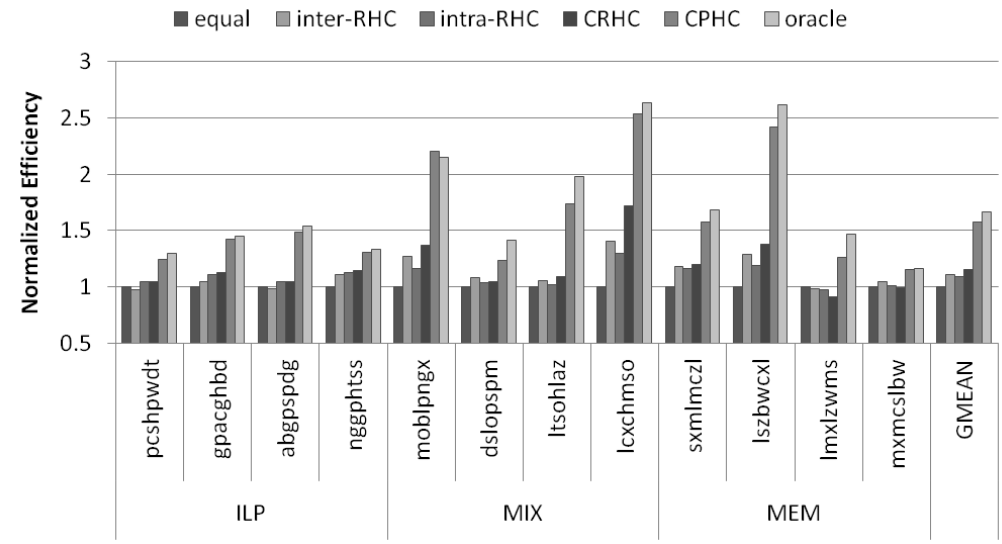
8.6.3 Performance & Efficiency

Figure 8.7(a) shows the comparison of the weighted speedups between different resource allocation policies. As expected, equal partition policy usually yields lowest weighted speedup among all the policies investigated in this dissertation. Inter-RHC and Intra-RHC improves the performance over equal partition policy as it dynamically adapts allocations for either inter-core or intra-core resources. CRHC further improves the weighted speedup, as it attempts to adjust the resource allocation on both inter-core and intra-core level. However, for some workloads, these reactive allocation policies may leads to inferior performance compared with equal partition. This is because they rely on the trial runs to explore the appropriate resource allocation, which means workloads may spend some trial runs in an inappropriate resource allocation. That also explains why these dynamic policies only have a small improvement over the equal partition policy. Our proposed predictive hill-climbing scheme avoids trial runs, and achieves an average of 11.6% over the baseline scheme and 9.3% over the CRHC scheme. In general, CPHC yields higher speedup in MIX workloads because in such workloads, the resource requirements of the programs are more diversified, resulting in higher potential for resource management. Compared with the Oracle scheme, the CPHC has approximately 3% less speedup. This is contributed by: (a) the imperfection of the performance model;(b) the lack of future knowledge; (c) hill-climbing being trapped in local optima.

Figure 8.7(b) further shows the efficiency improvements for different resource allocation policies. It is observed that CPHC has an average efficiency improvement



(a) Improvement in Weighted Speedup



(b) Efficiency Improvement

Figure 8.7: Performance and efficiency comparison for different resource management policies. GMEAN refers to Geometric Mean.

of 57.4% over the baseline, and 36.5% over CRHC.

8.6.4 QoS Enforcement

The QoS target is defined as the target IPC relative to the alone-execution IPC, expressed in the form of percentages [7][56]. The proposed resource management framework can convert this QoS target into resource usage requirements [7], thereby enforce QoS for an application by regulating the amount of allocated resources. The quality of such QoS enforcement is demonstrated in Figure 8.8, where for each workload, only one program is enforced with the QoS targets and the remaining programs do not have QoS objectives. As one can see, the relative IPCs of the programs keep a good track of the QoS targets. For some programs, such as *povray*, *gcc*, and *astar*, the relative IPC at the 20% QoS target is significantly off the target. This is because even with the minimum allocation on each resource, the

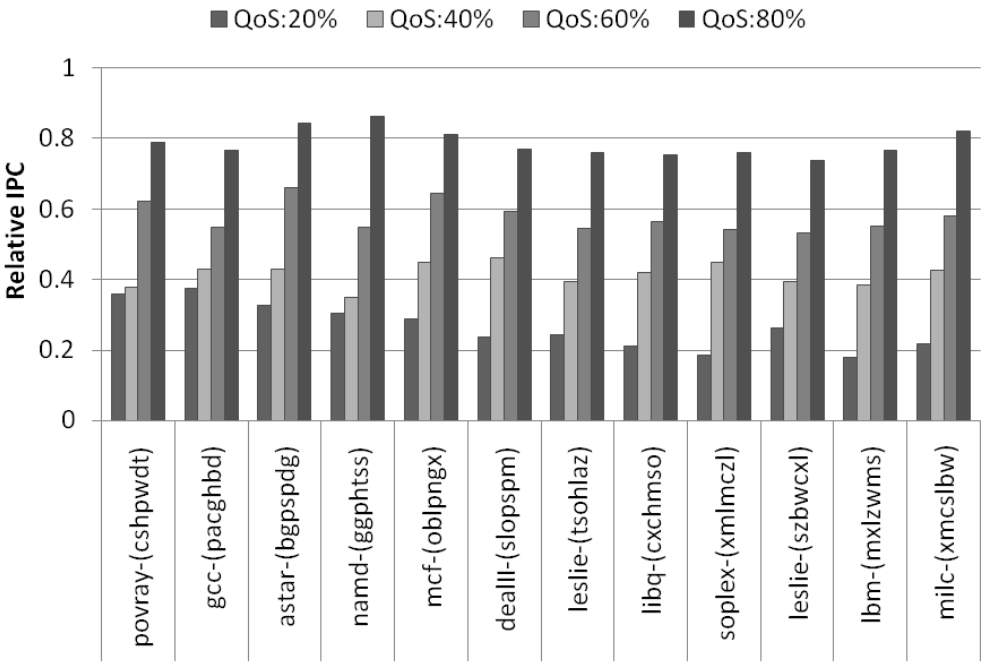


Figure 8.8: QoS targets enforcement.

relative performance of these programs are still much larger than 20%. Hence, such QoS target is *ill-suited* for these programs. Overall, it is observed that the proposed framework could enforce QoS within 6.1% for 80% target, 6.7% for 60% target, and 5.9% for 40% target. Hence, this framework is suitable for the enforcement of elastic QoS objective [7].

8.7 Summary

This chapter demonstrates that for a Chip Multiprocessors (CMP) supporting per-core Simultaneous Multithreading (SMT), both intra-core and inter-core resources need to be managed simultaneously in order to achieve high resource utilization and deliver controllable performance. Therefore, this chapter presents a predictive resource management framework that coordinates both inter-core and intra-core resources for throughput and QoS. This framework uses a set of hardware-efficient online profilers and an analytical performance model to predict the application's performance with different intra-core and/or inter-core resource allocations. Based on the predicted performance, the resource allocator identifies and enforces near optimum resource partitions for each epoch without any trial runs. The experimental results show that the proposed framework improves weighted speedup by an average of 11.6% compared with equal partition scheme, and 9.3% compared with the learning-based resource manager. This experiment also shows this framework enforces QoS targets within 6.7%. This predictive resource management framework offers a promising way to coordinate both inter-core and intra-core resources in CMPs.

Chapter 9

Conclusions and Future Research Directions

9.1 Conclusions

Power-efficient computing through core specialization has become increasingly important for Chip-Multiprocessors to alleviate the power density constraints. Single-ISA Heterogeneous Multi-core Processor (SHMP) emerges as an important and attractive form of core specialization as it avoids painful ramification of modifying the compilers and applications, yet still provides the core-level heterogeneity to meet the diverse requirements of the workloads. However, to unleash the full potential of SHMP, the workload heterogeneity has to be efficiently and accurately translated to the hardware heterogeneity, which is challenging and remains an open problem.

This dissertation proposes and evaluates a comprehensive solution to this problem by leveraging an analytical performance model as the basis to bridge the gap between workload heterogeneity and hardware heterogeneity. The proposed solution covers the off-line program resource demand analysis and off-line program-core mapping, and the on-line heterogeneity-aware application scheduling and dynamic multiple resource management. In each of these aspects, the proposed solution shows significant improvement over the state-of-the-art in terms of energy efficiency, throughput, and scalability.

Chapter 5 presents an off-line framework to analyze the program resource

demand by using program inherent characteristics and an augmented analytical performance model. This chapter proposes an off-line modeling technique for memory level parallelism, which decouples the analytical performance model from partial cache simulations, accelerating the speed and efficiency of the resource demand estimation. This proposed framework, as a stand alone technique, is useful in early stage design space exploration and workload capacity planning. It also lays the foundation for the techniques that rely on resource demands to further close the gap between workload diversity and core-level heterogeneity. Chapter 6 demonstrates such a framework that matches the estimated resource demands with the appropriate cores in static SHMPs. The combination of these two techniques gives a complete solution for off-line program scheduling in static SHMPs.

Chapter 7 shows the analytical performance model could also be applied on-line to predict the performance of an application on different cores, and guide the application scheduling in static SHMPs. With only a few kilobytes of hardware cost for the on-line profilers, the proposed predictive scheduler completely eliminates the trial runs needed by existing heterogeneity-aware scheduler, and improves the system throughput by an average of 11.2% and efficiency by an average of 36.2% compared with the state-of-the-art research scheduler.

Finally, this dissertation proposes to leverage the analytical performance model for managing multiple interacting resources online in dynamic SHMPs. With a set of hardware-efficient online profilers, the proposed management framework is able to predict the application's performance with different intra-core and/or inter-core resource allocations, and thereby pro-actively enforce the resource allocations to meet system performance objectives.

Overall, the proposed off-line and on-line techniques as a whole constitutes a comprehensive, efficient, cost-effective solution to unleash the full potential of efficient heterogeneous computing.

9.2 Future Research Directions

The proposed techniques for efficient heterogeneous computing in this dissertation can be further extended in the following directions:

9.2.1 Improving the Efficiency of On-line Profilers

While the hardware cost of the on-line profilers proposed in this dissertation is much less hardware than that of the machine learning model based on Artificial Neural Networks (ANN), it is not negligible and is expected to increase as the reconfigurable resource becomes more fine-grain. Therefore, more cost-effective on-line profiling mechanism is needed. The search of hardware saving techniques can be steered in two directions. One of them is to reduce the size of histograms by taking advantage of the locality of inherent program characteristics. For example, in most cases, the instruction dependency chain length does not exceed half of the ROB size; hence, the critical dependency chain histogram only need to have half of the ROB entries without losing modeling accuracy. The other direction is to use empirical knowledge to *derive* the some of the the statistics rather than use profilers to *extract* them during the execution of the application. For example, the ROB size and the amount of exposed ILP has been empirically demonstrated to follow the power law relationship [12], which means n times the size of ROB can lead to approximately $\sqrt{n}$ times the amount of exposed ILP. Therefore, it is possible to profile the critical dependence chain length for one specific ROB size, and use the power law relation-

ship to derive the amount ILP for different ROB sizes. In this way, only one critical dependence chain histogram is required for ILP profiling. However, the empirical estimation could inevitably compromise the accuracy of the performance modeling, which in turn may undermine the end performance of the resource management scheme. Hence, one has to carefully balance the trade-offs between the hardware cost and the model accuracy.

9.2.2 Expanding the Types of Heterogeneous Resources

The MLP modeling technique presented in the dissertation only captures the demand L2 misses, and does not consider the impact of memory prefetchers. For a CMP system with aggressive hardware prefetching, the default off-line stack distance model is insufficient to accurately estimate the number of L2 load misses. Therefore, the model needs to be augmented with the capability to recognize the prefetching patterns and replay them off-line. The on-line implementation of the stack distance model also needs to be modified so that the prefetching requests can also update the LRU stack.

In addition, the analytical model used in this dissertation assumes that the processor core, no matter how simple it is, is an out-of-order superscalar processor. In practice, the single-ISA heterogeneous multicore may contain a mixture of in-order cores and out-of-order cores. While the interval analysis theory still holds for in-order cores, dramatic changes in the ILP modeling is required to estimate the performance of in-order cores. Therefore, in the presence of both in-order and out-of-order cores, two separate performance models are needed so that the performances on these two execution styles can be predicted.

Bibliography

[1] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pp. 23–32, 2006.

[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 81–92, 2003.

[3] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.

[4] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proceedings of 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 381–394, Dec. 2007.

[5] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th International Symposium on Computer Architectures*, pp. 186–197, June 2007.

[6] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 29–40, 2006.

[7] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proceedings of the 40th International Symposium on Microarchitecture*, pp. 343–355, 2007.

[8] P. Joseph, K. Vaswani, and M. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 99–108, 2006.

[9] B. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pp. 340 –351, 2007.

[10] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 195–206, 2006.

[11] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Design Automation Conference*, pp. 927–930, 2009.

[12] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 338–349, June 2004.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 1–37, 2009.

[14] H. Hofstee, "Power efficient processor architecture and the CELL processor," in *Proceedings of the 11th High Performance Computer Architecture*, pp. 258–262, Feb. 2005.

[15] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th Design Automation Conference*, pp. 338–342, 2003.

[16] "International technology roadmap for semiconductors," in *http://public.itrs.net*, 2006.

[17] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *Proceedings of the 21st International Conference on Computer Design*, pp. 481–488, 2003.

[18] J. Chen and L. K. John, "Energy aware program scheduling in a heterogeneous multi-core system," in *Proceedings of 2008 IEEE International Symposium on Workload Characterization*, pp. 1–9, 2008.

[19] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proceeddings of the 11th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 175–184, 2006.

[20] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 233–244, 2002.

[21] S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing," in *Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 239–251, 2006.

[22] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in SMT processors," in *Proceedings of the 37th International Symposium on Microarchitecture*, pp. 171–182, 2004.

[23] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 191–202, 1996.

[24] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 318–327, 2001.

[25] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th International Symposium on Microarchitecture*, pp. 423–432, 2006.

[26] D. Kaseridis, J. Stuecheli, J. Chen, and L. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems," in *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, pp. 1–11, 2010.

[27] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proceedings of the 41st International Symposium on Microarchitecture*, pp. 318–329, 2008.

[28] T. S. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: an analytical approach," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 402–411, 2007.

[29] B. C. Lee and D. Brooks, "Efficiency trends and limits from comprehensive microarchitectural adaptivity," in *Proceedings of the 13th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 36–47, 2008.

[30] P. Bose and T. M. Conte, "Performance analysis and its impact on design," *Computer*, vol. 31, pp. 41–49, May 1998.

[31] R. L. Mattson, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[32] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in SMT processors," in *Proceeding of the 14th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 133–144, 2009.

[33] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, pp. 59–67, Feb. 2002.

[34] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," *HP Technical Reports*, 2008.

[35] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 83–94, 2000.

[36] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, pp. 50–58, 2 2002.

[37] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[38] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: a power-performance simulator for interconnection networks," in *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 294–305, 2002.

[39] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 128–138, 2000.

[40] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 63–74, 2008.

[41] "Spec cpu2006 benchmark suit," in *http://www.spec.org*.

[42] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, vol. 7, pp. 1–28, 2005.

[43] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 338–349, 2007.

[44] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger, "Multitasking workload scheduling on flexible core chip multiprocessors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 187–196, Oct. 2008.

[45] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the 9th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 234–244, 2000.

[46] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE Micro*, vol. 20, no. 6, pp. 26–44, 2000.

[47] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 57–68, 2007.

[48] B. Fields, S. Rubin, and R. Bodık, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 74–85, 2001.

[49] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, 1993.

[50] M. Haungs, P. Sallee, and M. Farrens, "Branch transition rate: a new metric for improved branch classification analysis," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pp. 241 –250, 2000.

[51] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John, "Measuring program similarity: Experiments with SPEC CPU benchmark suites," in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 10–20, 2005.

[52] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic," in *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 204–213, 2001.

[53] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pp. 167–178, 2006.

[54] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide*.

[55] P. B. Daniel and M. Cesati, *Understanding the Linux Kernel, Third Edition*, ch. 7. O'Reilly Media, 2005.

[56] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero, "Predictable performance in SMT processors," in *Proceedings of the 1st Conference on Computing Frontiers*, pp. 433–443, 2004.

[57] M. Gerndt, *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, Bonn, Germany, Dec. 1989.

[58] D. E. Knuth, *The TEXbook*. Reading, Mass.: Addison-Wesley, 1984.

[59] F. A. Bower, D. J. Sorin, and L. P. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," *IEEE Micro*, vol. 28, pp. 17–25, May 2008.

[60] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *IEEE Computer*, vol. 38, pp. 32–38, Nov. 2005.

[61] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *Journal of Instruction Level Parallelism*, vol. 5, pp. 1–33, Nov. 2003.

[62] M. Maheswaran and H. J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems," in *Proceedings of the Heterogeneous Computing Workshop*, pp. 57–69, June 1998.

[63] C. S. Ballapuram, A. Sharif, and H.-H. S. Lee, "Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors," in *Proceedings of the 31th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, pp. 60–69, Mar. 2008.

[64] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi, "Power-aware scheduling under timing constraints for mission-critical embedded systems," in *Proceedings of the 38th Conference on Design Automation*, pp. 29–40, 2001.

[65] D. Burger and T. M. Austin, "The simplescalar tool set version 3.02," in *http://www.simplescalar.com*.

[66] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 363–374, 2008.

[67] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proceedings of the 36th International Symposium on Computer Architecture*, pp. 93–104, 2009.

[68] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," *Proceedings of 31st Annual International Symposium on Computer Architecture*, pp. 64 – 75, 2004.

[69] D. Williams, A. Sanyal, D. Upton, J. Mars, S. Ghosh, and K. Hazelwood, "A cross-layer approach to heterogeneity and reliability," in *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign*, pp. 88–97, 2009.

[70] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proceedings of 36th Annual International Symposium on Computer Architecture*, pp. 302–313, 2009.

[71] G. Yan, X. Liang, Y. Han, and X. Li, "Leveraging the core-level complementary effects of PVT variations to reduce timing emergencies in multi-core processors," in *Proceedings of the 37th International Symposium on Computer architecture*, pp. 485–496, 2010.

[72] K. A. Bowman, A. R. Alameldeen, S. T. Srinivasan, and C. B. Wilkerson, "Impact of die-to-die and within-die parameter variations on the clock frequency and throughput of multi-core processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 12, pp. 1679–1690, 2009.

[73] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[74] E. O. Brigham, *The Fast Fourier Transform*, ch. 13. Prentice-Hall, 1974.

[75] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Exploiting structural duplication for lifetime reliability enhancement," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 520–531, 2005.

[76] H. Wang, M. Miranda, W. Dehaene, F. Catthoor, and K. Maex, "Systematic analysis of energy and delay impact of very deep submicron process variability effects in embedded SRAM modules," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 914–919, 2005.

[77] S. Ghiasi, T. Keller, and F. Rawson, "Scheduling for heterogeneous processors in server systems," in *Proceedings of the 2nd Conference on Computing Frontiers*, pp. 199–210, 2005.

[78] D. A. Jimenez, "Piecewise linear branch prediction," in *Proceedings of the 32nd annual International Symposium on Computer Architecture*, pp. 382–393, 2005.

[79] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark, "Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 314–323, 2003.

[80] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, 1991.

[81] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt, "Branch classification: a new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 22–31, 1994.

[82] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: what makes two-level branch predictors work," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 52–61, 1998.

[83] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 190–200, 2005.

[84] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi

[85] L. Porter and D. M. Tullsen, "Creating artificial global history to improve branch prediction accuracy," in *Proceedings of the 23rd International Conference on Supercomputing*, pp. 266–275, 2009.

[86] S. Mantripragada and A. Nicolau, "Using profiling to reduce branch misprediction costs on a dynamically scheduled processor," in *Proceedings of the 14th International Conference on Supercomputing*, pp. 206–214, 2000.

[87] R. H. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the 19th Annual International Conference on Supercomputing*, pp. 111–120, 2005.

[88] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai, "The impact of if-conversion and branch prediction on program execution on the intel® itanium processor," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 182–191, 2001.

[89] H. Kim, M. A. Suleman, O. Mutlu, and Y. N. Patt, "2D-profiling: Detecting input-dependent branches with a single input data set," *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization*, vol. 0, pp. 159–172, 2006.

[90] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," *IEEE/ACM International Symposium on Microarchitecture*, vol. 0, pp. 43–54, 2005.

[91] S. P. Kim and G. S. Tyson, "Analyzing the working set characteristics of branch execution," in *Proceedings of the 31st ACM/IEEE international symposium on Microarchitecture*, pp. 49–58, 1998.

[92] A. Joshi, L. Eeckhout, L. John, and C. Isen, "Automated microprocessor stressmark generation," in *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pp. 229 –239, 2008.

[93] Y. Sazeides, A. Moustakas, K. Constantinides, and M. Kleanthous, "The significance of affectors and affectees correlations for branch prediction," in *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pp. 243–257, 2008.

[94] M. U. Farooq, L. John, and J. M. F., "Compiler controlled speculation for power aware ILP extraction in dataflow architectures," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pp. 324–338, 2009.

[95] B. Simon, B. Calder, and J. Ferrante, "Incorporating predicate information into branch predictors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 53–64, 2003.

[96] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, 2001.

[97] J. Casazza, "White paper first tick, now tock: Intel microarchitecture (Nehalem)," 2009.

[98] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of the 39th International Symposium on Microarchitecture*, pp. 347–358, 2006.